Marie Skłodowska-Curie Actions

Grant agreement ID 101031688

PLATON - Platform-aware LArge-scale Time-Series prOcessiNg

Deliverable D2.1

Report on the Multi-Node Data Series Index

Edited by Panagiota Fatourou

List of Authors

Panagiota Fatourou
Eleftherios Kosmas
Themis Palpanas
Emmanuel Papadospyridakis

Code available at
https://github.com/ConcurrentDistributedLab/DRESS

September 2022

TABLE OF CONTENTS

LIST OF PAPERS

- Panagiota Fatourou, Eleftherios Kosmas, Themis Palpanas, Emmanuel Papadospyridakis, ``Data Series Indexing Techniques for a Multi-Node Environment,'' FORTH ICS TR 479, April 2022.
- Karima Echihabi, *Panagiota Fatourou*, Kostas Zoumpatianos, Themis Palpanas, Houda Benbrahim, ``Hercules Against Data Series Similarity Search,'' submitted for publication. LIPADE-TR-No 7, October 2022.

SUMMARY

PLATON aimed at exploiting the full computational power of modern architectures and multi-node configurations. This required to develop new algorithms and techniques for highly-efficient data series processing in a multi-node setting. This encompasses the design and implementation of low-cost data partitioning and mapping techniques for answering queries on large collections of data series and load balancing algorithms and primitives for multi-node query processing that will result in much better performance, high scalability, and fault-tolerance in large-scale data series processing.

In this work, we study the problem of efficiently executing exact similarity search queries on multi-node environments, while still exploiting the computation power of the multiple cores supported on each node. We build upon MESSI [1], a state-of-the-art in-memory data series index. We provide techniques to partition the data series to the different nodes and cope with workload imbalance problems. Our experimental evaluation reveals that the resulting implementations achieve speedup that is close to the ideal compared to MESSI, on a distributed system with a small number of nodes.

1. Introduction

Nowadays applications across diverse domains (such as finance, environmental science engineering and other [12], [28]) produce large and complex data, usually in the form of data series[1] [4], [8], [9], [10], that need to be processed and analyzed. The most common type of queries that different applications need to answer on such complex and massive sets of data series is *similarity search*. Similarity search is the basis to perform many data mining tasks such as *classification* or *clustering* [13], [14], [15].

An application of interest concerns Airbus[2], which currently stores petabytes of data series, describing the behavior over time of various aircraft components (e.g., the vibrations of the bearings in the engines), as well as that of pilots (e.g., the way they maneuver the plane through the fly-by-wire system) [4]. Experts need to run various data analytics algorithms on these data efficiently. To achieve this, in many cases these algorithms operate only on a subset of the data (e.g., the data relevant to landings from Air France pilots) that fits in memory. For in memory data analytics, to perform fast complex data analytic operations (e.g., searching for similar patterns), in-memory data series indices are built to allow search over a large amount of data series during query processing to be performed efficiently. In these cases, the performance of both the index creation phase and the query answering phase is crucial.

Recently, researchers, MESSI[1], Paris+[33][34], SING[32], focused on improving the performance of both the index creation and the query answering phases, by incorporating parallelism. MESSI is a state-of-the-art in-memory data series index for answering similarity search queries on big collections of in-memory data series. It is designed to accelerate both index construction and similarity search processing times. Specifically, it introduces several parallelization techniques for multi-core architectures (and their SIMD support) that concurrently (and thus, effectively) execute the calculations required for index construction and query answering. However, notice that in-memory approaches, like MESSI, impose limitations on the volume of data series that can be processed, since this dataset must fit into main memory.

On the other hand, the greediness of modern applications, in whatever regards data series production, turns even state-of-the-art indexing techniques (e.g., MESSI) inadequate. Additionally, the need for faster data analytics is permanent. To achieve both better scalability (on dataset size) and response time (while answering queries), an idea is to overcome the limitations imposed by the capabilities of a single computation node by incorporating multiple nodes and designing a distributed solution. This way, the full computational power of both modern architectures and multi-node configurations can be exploited. Specifically, using a multi-node environment and applying appropriate multi-node (i.e., distributed) and multi-core (i.e., in a single node) techniques, the indexing and mining of large

[1] A *data series* is an ordered sequence of data points. When the ordering dimension is time, we can call it *time series*.
[2] https://www.airbus.com/

volumes of data series can be done faster, when compared to centralized approaches (e.g., MESSI). This is the focus of the current paper. Our contribution is presented in Section 1.1.

Interestingly, following this direction, a distributed multi-node data series index construction and query answering algorithm [31], namely DPiSAX, has been recently proposed. DPiSAX is a parallel index construction algorithm that takes advantage of distributed environments to efficiently build iSAX-based[3] indices over very large volumes of data series and answer both approximate and exact similarity queries, in a distributed fashion. Specifically, the basic idea of DPiSAX is to parallelize the workload by executing several similarity searches queries in parallel. To better enhance this, it starts by partitioning the dataset so that each node contains closely related data series, on which they build indexes. Then, during query answering phase the queries are separated in a similar way, so that each node receives those that are in close distance to the part of the dataset it maintains. This is enough for DPiSAX to answer approximate answers. To answer an exact similarity query, the previously discovered approximate answer is propagated to all other nodes and it is used to discover the exact answer.

1.1 Contribution

In this paper, we study multi-node indexing techniques that exploit the full computational power of a multi-node environment. We introduce *DRESS* an in-memory multi-node data series index that can answer similarity search queries in a highly efficient way. It achieves this by executing an instance of a variant of MESSI on each node of the system. Recall that MESSI is an in-memory data series index that it effectively uses multi-core architectures to concurrently execute the computations needed for both index construction and query answering. Briefly, MESSI starts by converting the data series into iSAX summarizations[3] and builds a tree index for them. During this process, it takes advantage of multithreading techniques. Afterwards, to answer a similarity search query, MESSI calculates the iSAX summarization of this query and uses it to efficiently traverse the tree index, by reducing both the number of elements visited and the number of similarity calculations it performs. We present more details about MESSI in Section 3.

Since DRESS is based on MESSI, it inherits MESSI's nice performance properties on each node, while it exploits the full computational power of a multi-node environment. DRESS appropriately coordinates MESSI instances to exploit the computational power of multiple nodes. We have run DRESS in a system with two nodes and at this setting, it achieves a speedup of up to 1.8 against MESSI, which is close to the ideal. To achieve this, we had to go through many steps and build upon several techniques. A small summary of the steps we went through, follows below.

We start by splitting the dataset into N chunks, where N is the number of nodes in the distributed environment and having each instance of MESSI (running on some node) work on a single one of them. However, for some of the datasets we experimented with, this simple approach fails even to

[3] *Indexable symbolic aggregate approximation (iSAX) is a* form of summarization over a data series.

achieve speedup; on the contrary, the system suffers from performance degradation due to workload imbalance. To achieve better workload balance, we next attempt to shuffle the initial dataset, so that each node takes a chunk that probably contains data series following a more uniform distribution, than the corresponding chunk of the initial dataset. Our experiments reveal that in some datasets (e.g., in Seismic), shuffling may not solve the workload imbalance. Specifically, we observe some cases where queries were in close distance to only a few data series on the whole initial dataset. As a result, the node that takes the chunk of the shuffled dataset containing these data series, has better pruning degree than the other node, who performed calculations on data series that are not in close distance to the query. Nevertheless, with the shuffling technique, we achieve better performance on query answering rather than the naïve approach. The reason for this is that both nodes take some of the aforementioned data series that are in close distance to the queries. This makes query answering based on shuffling to perform better, due to the calculation of fewer distances. But still, the performance of the query answering is worse than in MESSI.

To discover the cause of the workload imbalance that was responsible for the observed performance degradation, we had to study the details of MESSI and the techniques it employs. Our study resulted in simple multi-node techniques that extend MESSI and achieve better workload balance.

As we discuss in Section 3, during query answering, MESSI uses a shared variable called *best-so-far* (*BSF*) that contains the best discovered distance value between the data series and the query. MESSI takes advantage of BSF to examine only a subset of the root subtrees of the tree index and *prune* the rest of them. The first technique we study, called *system-wide best-so-far (SW-BSF)*, shares the computed BSF values among the nodes to avoid situations where a node performs useless calculations on subtrees that could have been pruned using the BSF value of some other node. Briefly, the basic idea is to require from each node that discovers a new value for BSF to broadcast it to all other nodes. By doing so, each node updates its local BSF using the best value discovered for BSF across all the nodes. This emulates in a distributed fashion, the way that the threads of MESSI communicate with each other the updates on BSF. Using this technique, DRESS achieves speedup (over the single-node MESSI) for all the datasets we experimented with, since it achieves better workload balance.

However, there are cases where the SW-BSF technique may not work well enough. For instance, consider a scenario with N=2, where the dataset is split into two equally sized parts and assume that all the data series of the first node are similar to a specific query Q, while none of the data series of the second node is similar with Q. In this case, during the query answering phase, the BSF calculated by the first node is always better than the BSF calculated by the second node. So, after the first node sends its BSF to the second, the latter prunes most of its tree index elements and thus, may complete its work much faster than the first one. In this case, the second node remains idle for a long period of time, waiting for the first node to complete its work. This results again in workload imbalance. To avoid such scenarios, we follow a *work-stealing multi-node approach*. Considering the previous scenario, this approach will allow the second node to undertake part of the first node's work. Thus,

the second node does not remain idle, and the work of the first node is completed faster (with the help of the second node).

We remark that the methodology we followed above constitutes the main contribution of this paper. Notice that the above techniques exploit the increased computational power of multiple nodes to enhance the performance of DRESS when answering queries one after the other. Specifically, all the nodes are taking part in the query answering procedure of each query. Going a step further, we studied a simple technique where the queries are partitioned among the nodes, so that each query is answered only by a single node. By doing this, multiple queries are answered in parallel.

Finally, we implemented and experimentally evaluated each one of the techniques we introduced for DRESS in a system with two nodes. We compared our implementations against MESSI using datasets that contain both synthetic and real data series. DRESS achieves 1.8x speedup against MESSI.

1.2 Roadmap

In Section 2, we present some basic definitions, and we describe the required terminology. In Section 3, we provide a brief description of the techniques employed by the MESSI algorithm. In Section 4, we present in detail the techniques employed by DRESS and Section 5 provides both the complete pseudocode of DRESS and its implementation details. In Section 6, we present our experimental evaluation, where we discuss in detail the results of our experimental analysis. Finally, in Section 7, we summarize our results and present future wok directions.

2. Definitions & Preliminary Material

[Data Series] Data series or data sequences can be found in diverse domains, such as medicine, astronomy, and neuroscience. A *data series* S is an ordered sequence $< v_1, t_1 >, \dots, < v_n, t_n >$ of data points, where v_i is the value and t_i is the position of the i-th data point. If the ordering dimension is time, then S is a time series where the position t_i of the i-th data point is a point in time, and these points are ordered, i.e., $t_i < t_{i+1} < t_i$. If a data series S has n points, the length of S is equal to n.

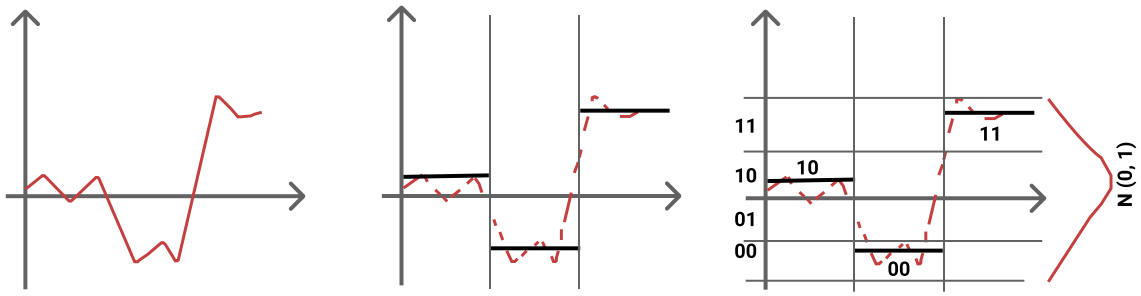
[Distance Measures] A distance measure reflects the underlying (dis)similarity between two data series. Common distance measures are the *Euclidian distance* (ED) [2] and the *dynamic time wrapping* (DTW) [3]. In this work, we focus on Euclidian distance, which is used by most existing data series indexes, like MESSI [1], ADS [5], PARIS [6], and PARIS+ [7]. More specifically, considering two data series $S_1 = (x_1, \dots, x_n)$ and $S_2 = (y_1, \dots, y_n)$, the Euclidian distance is defined as follows:

$$ED(S_1, S_2) = \sqrt[2]{\sum_{i=1}^{n} (x_i - y_i)^2}$$

[Similarity Search] Similarity search is one of the main problems in the data series analysis. Analysts perform a wide range of data mining tasks on data series in diverse domains. Existing applications for executing these tasks rely on performing fast similarity searches across big collections of data series. To achieve this, a lot of existing algorithms that solve the *approximate similarity search* problem works as follow, given a query data series and a collection of data series they return the data series in the collection that is in a close distance (sometimes with some guarantees) to the query series. This approximate answer lacks precision, which sometimes may harm mining results. In this work, we study the *exact similarity search* problem (also known as the nearest neighbor query) that returns the best existing answer (i.e., the nearest neighbor from the query data series)**.** More specifically, given a query data series S_q and a data series collection X, we want to identify the data series $S_a \in X$ that has the smallest distance to S_q among all the series in X. Note that efficiently performing nearest neighbor queries is more challenging than performing approximate similarity search queries.

[Data Series Summarization] To efficiently execute nearest neighbor queries on a large collection of data series, we focus on solutions that are built upon indexes. Specifically, to further optimize searches on data series collection, the index is built usually using summarizations of the data series.

[Piecewise Aggregate Approximation] We now provide more details on the way a data series can be summarized. The *Piecewise Aggregate Approximation* (*PPA*) representation divides a data series S into segments of equal length, and for each such segment, it computes the mean value of the points of each of these segments; all these mean values, constitute the PAA representation of S. As an example, consider the (raw) data series depicted in Figure 1-a. Its PPA representation with three segments is shown in Figure 1-b.



(a) raw data series (b) PAA representation (c) iSAX representation

Figure 1 - The iSAX Representation

[Indexable symbolic aggregate approximation] The *indexable symbolic aggregate approximation (iSAX)* of a data series, is defined based on its PPA representation. The iSAX summarization technique divides the y-axis space into different regions and assigns a bitwise symbol to each region. Then, each segment *w* of the data series PPA representation is assigned the symbol of the y-axis region in which it falls into. As an example, the iSAX representation for the data series depicted in Figure 1 is 10|00|11, as shown in Figure 1-c.

[Lower Bound Distance and Real Distance Calculation] Given a query data series the iSAX summary helps us to check if the iSAX summary of the query is similar to the iSAX summary of each data series in the collection. Well-known indexes based on iSAX summaries perform two types of calculations to efficiently achieve this, which we describe below. Given two data series, their *lower bound distance* is the Euclidian distance between the iSAX summaries of the data series.

If this distance is lower than a threshold the data series are dissimilar. The real distance between S_1 and S_2 is the actual Euclidean distance between them. Every time we computed this actual distance between two data series, we say that we perform a real distance calculation. Note that, the lower bound distance between two data series, is not as precise as the real distance, as it is based on summarizations of the data series. If the lower bound distance, between two data series, is larger than a threshold then it is sure that the data series are dissimilar so there is no need to calculate the real distance.

[Distributed Message-Passing System] A *distributed message-passing system* is a system whose components (often referred to as *nodes*), are located on different computers. Each node of the system is an independent multicore machine with its memory and disk storage. Thus, each node supports the execution of multiple threads. Threads located in different nodes communicate and coordinate by passing messages to one another. We consider a system whose node supports *SIMD* (*single instruction, multiple data*) computation, where SIMD refers to a parallel architecture that allows the execution of the same operation on multiple data simultaneously. Threads of the same node, communicate through accessing shared variables.

3. MESSI - In-Memory Data series Indexing

The proposed implementations are built on top of MESSI [1], an in-memory data series index. In this section, we motivate our choice of MESSI, and we are providing some details on how MESSI supports query answering.

The state-of-the-art techniques failed to deliver the time performance required for the interactive exploration or analysis of large data series collections. MESSI is a state-of-the-art in-memory data series index, recently proposed in [1]. MESSI was designed to accelerate both index construction and similarity search processing times. It effectively uses multi-core architectures and the SIMD hardware they provide, to concurrently execute the computations needed for both index construction and query answering.

3.1 Overview

MESSI is comprised of two main phases, the indexing, and the query answering phase. In the *indexing phase*, MESSI builds a tree index. As the volume of data is large the tree index is constructed based on the data series iSAX summaries. MESSI first calculates all the data series iSAX summarizations in parallel and stores these summaries into a set of iSAX buffers. Data series that have similar summarization are placed into the same iSAX buffer. Subsequently, the data series of each of these buffers will be stored into each of the subtrees of the constructed tree index. This technique allows MESSI to build the tree index in an embarrassingly parallel way.

To answer a query, MESSI first calculates the iSAX summary of the query. Subsequently, it traverses the index tree to find the most appropriate data series based on the iSAX summary lower bound distances. The distance of these data series from the query series is stored in a variable called *best-so-far* (*BSF*). Then, the index tree is traversed concurrently to search for relevant data series (i.e., for data series whose iSAX summary distance from the query summary is smaller than the BSF). For each node in the tree index, MESSI checks whether the lower bound distance between the query and the iSAX summary of the node is higher than the current value of the BSF. If this is the case the subtree emanated from this node is pruned. Otherwise, the node is placed in one of the priority queues that MESSI stores data series that need to be further examined; moreover, the BSF is updated appropriately. Finally, MESSI processes in parallel the nodes from the priority queues. For each node, it calculates the lower bound distance and if it is still lower than the BSF it calculates the real distances between the data series, the node contains, and the query series, and updates BSF if need it. Figure 2 depicts the two phases described above.

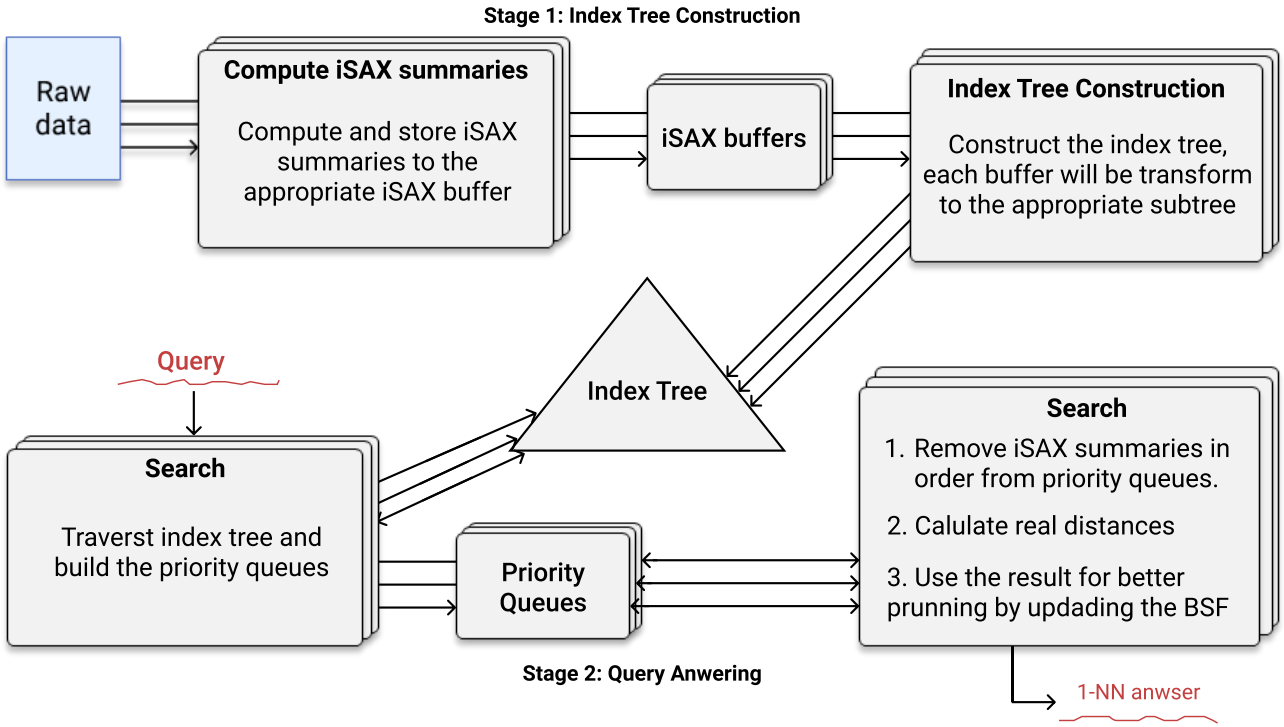


Figure 2 - MESSI index construction and query answering

3.2 Indexing Phase

We now provide more details on the index tree construction phase which is depicted in Figure 3. MESSI first loads the raw data from the disk to the main memory, into an array called RawData. This array is spliced into a predetermined number of chunks, so each worker thread can operate independently from the others to calculate the iSAX summaries of the data series in the chunks assigned to it. Chunks are assigned to index workers one after the other using Fetch & Increment. When an index worker calculates an iSAX summary it stores it into a buffer chosen appropriately from the collection of iSAX buffers. Each iSAX buffer contains data series that will be stored in the same subtree of the tree index root. To reduce synchronization between threads when accessing the iSAX buffers, each iSAX buffer is split into as many parts as the workers, and each worker stores iSAX summaries into the part of the iSAX buffer that is assigned to it. The number of iSAX buffers is at most 2^w where w is the number of segments in the iSAX summary representation of each data series (w is fixed to 16 in the MESSI paper).

When the iSAX summaries for all the raw data series have been computed, the index workers proceed in the construction of the tree index. Using a fetch & increment object, each worker is assigned an iSAX buffer to work on and creates the corresponding index subtree. Each index worker processes distinct subtrees. When an index worker finishes with the assigned iSAX buffer, it continues with the next available iSAX buffer, that has not yet been procced. When all the iSAX buffers have been

processed and the index tree has been fully built, this index tree can be used to answer similarity search queries. The index tree is comprised of three types of nodes:

1. The root node points to several children nodes, 2^w in the worst case, when the series in the collection cover all possible iSAX summaries.
2. Inner nodes that contain the iSAX summaries of all the series stored in the subtrees emanated from the node. Each inner node has exactly two children.

3. Leaf nodes, each of which contains pairs of the iSAX summary of data series and a pointer to each raw data in the RawData array.

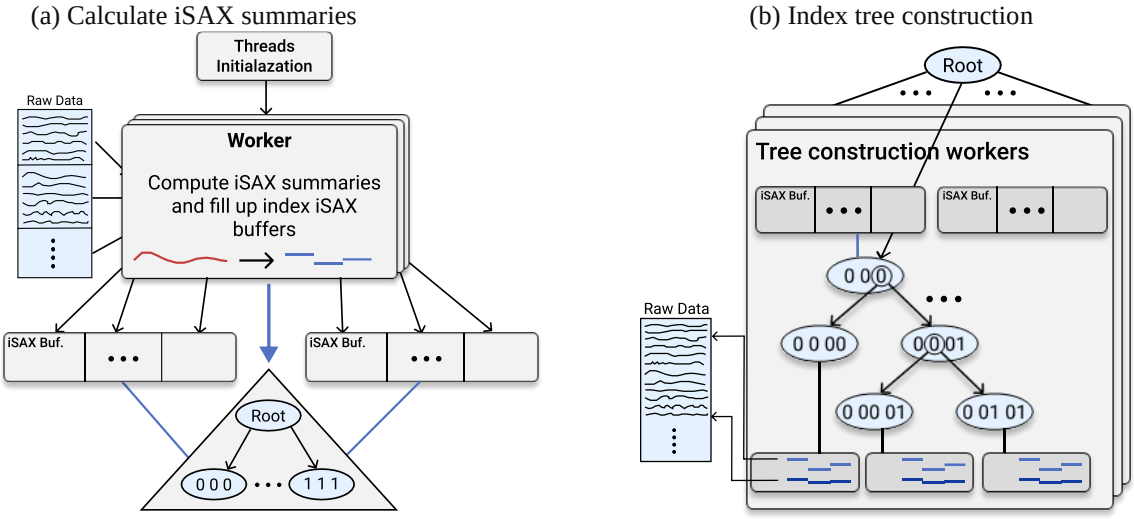
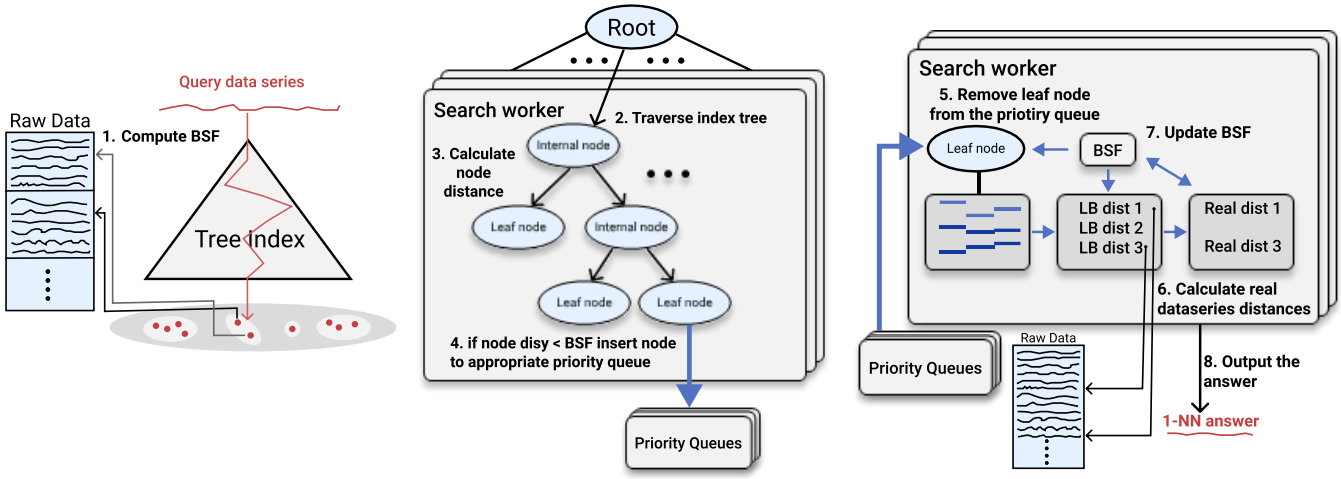


Figure 3 - Workflow and algorithms for MESSI index construction

3.3 Query Answering Phase

We next provide more details about the query answering phase which is depicted in Figure 4. To answer a query, MESSI first performs a search for the query iSAX summary in the index tree. This returns a leaf whose iSAX summary has the closest distance to the iSAX summary of the query. Then, MESSI calculates the real distance between the data series pointed by the elements of this leaf to the query series and stores the minimum of these distances into BSF. Subsequently, the index workers start traversing the index subtrees, using BSF to decide which subtrees will be pruned. The leaves of the subtrees that cannot be pruned are placed into a fixed number of priority queues, using the lower bound distance between the summaries of the query and each leaf node. To achieve load balancing when accessing the priority queues, each thread inserts elements in the priority queues in a round-robin fashion. When the entire tree has been traversed, each index thread chooses a priority queue to work on and repeatedly removes nodes from the priority queue to further examine them. Specifically, for each node, the thread first checks whether the lower bound distance of the leaf is larger than the current value of BSF. If it is so, the leaf is pruned. Otherwise, the worker thread needs to examine the

data series contained in the leaf. It does so, by first computing the lower bound distance between each of these series and the query, and if this distance is smaller than the BSF it also computes their real distance. During this process, we may discover a series, which has a smaller distance to the query than the current value of BSF. In this case, MESSI updates BSF. When a worker riches a leaf node, in a priority queue, whose distance is higher than the BSF it gives up this priority queue and starts working with the next non abandoned priority queue, as it is then ensured, that all the other elements in the abandoned queue have a higher distance to the query series. This process is repeated until all priority queues have been examined. During this process, the value of BSF is updated to always reflect the minimum distance seen so far. When this process completes, the data series which is more similar to the query is stored in BSF. Thus, MESSI returns the final value of BSF.



(a) Approximate search for calculating initial BSF

(b) Index tree traversal and node insertion into priority queues

(c) Node distance calculation from priority queues

Figure 4 – Workflow and algorithms for MESSI query answering

4. Distributed Data Series Processing (DRESS)

Although MESSI achieves very good data-series indexing and query processing speed, even for gigabytes of data, the greediness of modern applications, in whatever regards data series production, turns even state of the art indexing techniques (e.g., MESSI [1]) inadequate. To achieve higher performance and scalability one must exploit the full computational power not only of modern architectures but also of multi-node configurations. Ideally, if a dataset of size *S* is processed in time *T* using a single node, then when moving to a system with *N* nodes of similar power, we would like to be capable to answer queries on S in time $\frac{T}{N}$, thus achieving speedup N. However, in practice, this is hard to achieve, since i) workload imbalance issues may arise, ii) the total amount of work performed by the nodes may increase as we go from a system of one node to a system of more , and iii) excess communication overhead may be required. These issues can be addressed by appropriately partitioning the dataset and assigning parts of it to nodes, and by achieving a similar degree of pruning among nodes so that they perform more or less the same amount of work.

DRESS is an in-memory multi-node data series index that can answer similarity search queries in a highly efficient way. It achieves this by executing an instance of a variant of MESSI on each node of the system. Each node i) creates its tree index on a part of the dataset, ii) answers queries using its tree index, and iii) reports the data series closer to these queries, i.e., the *partial answers*, to a single *coordinator node*. More specifically, the coordinator node collects these partials answers and for each query, it reports the best of them, i.e., the *final answer* to the query.

DRESS appropriately coordinates MESSI instances to exploit the computational power of multiple nodes. Since it incorporates a variant of MESSI, DRESS inherits MESSI's nice performance properties on each node. We run DRESS in a system with two nodes and at this setting, it achieves a speedup of 1.8, which is close to the ideal. To do so, we had to go through many steps and build upon several techniques.

A detailed description of the techniques employed by DRESS is provided in the remainder of this section, incrementally, based on the difficulties we faced upon designing and implementing DRESS. A complete description of the DRESS algorithm and its configuration for our experimental evaluation is provided in Section 5. Briefly, in Section 4.1 we discuss some simple techniques that we experimented with in DRESS in order to support multiple working instances of MESSI on several nodes. These are data partitioning techniques that allow to each MESSI instance to work on each own part of the dataset. However, for some of the datasets we experimented with, these simple techniques resulted in a lack of speedup (or even in performance degradation) in the multi-node setting. Sections 4.2 and 4.3, present the DRESS approach which solves these problems.

4.1 Data Partitioning

Considering a multi-node system, achieving *workload balance* between nodes is of high importance. To do this, a fundamental problem to solve is how to appropriately *distribute* the dataset to nodes so

that all of them perform almost the same amount of work. A first *naïve approach* is to partition the dataset into N disjoint equally sized chunks (i.e., as many as the number of nodes) and assign one of them to each instance of MESSI (on each node) to work on (see Figure 5). For example, in a system with two nodes (i.e., N=2), namely n_1 and n_2, the dataset is split in half, resulting in two equally sized chunks, with the first chunk assigned to n_1 and the second chunk assigned to n_2. Then, each node runs an instance of MESSI using its assigned chunk.

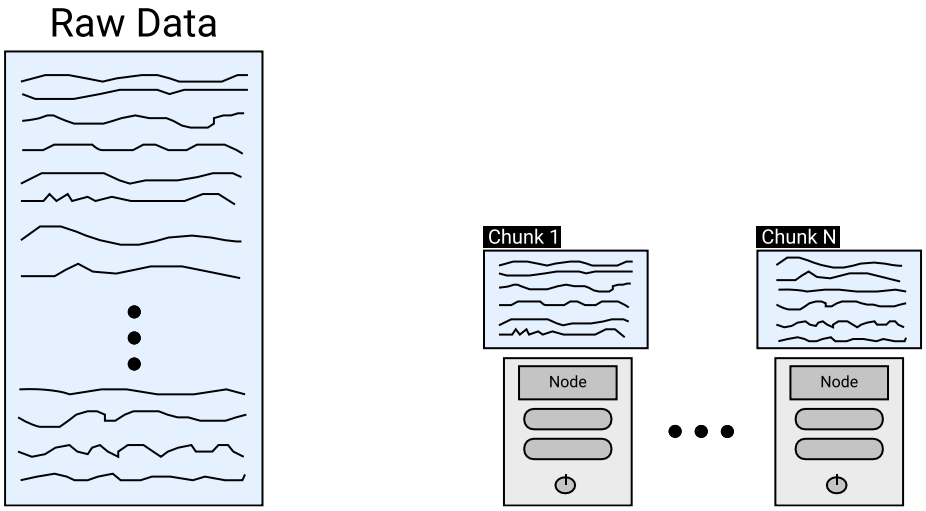


Figure 5 - Split dataset, naïve approach

However, this simple partitioning technique may result in a workload imbalance. For instance, considering again the previous example (where N=2) and a query Q, assume that most of the dataset's data series that are in close distance to Q reside in the first chunk, while only a few of them reside in the second chunk. In this case, upon query answering, n_1 may perform significantly less pruning than n_2 and thus, the priority queues of n_1 may be populated with a lot more items, than the priority queues of n_2. Therefore, n_2 may complete its work much faster than n_1 and remain idle for a long period (waiting n_1 to complete its work); if this occurs, it results in workload imbalance. To resolve these problems, we introduce two alternative techniques, namely shuffling of dataset and work-stealing, which are described in the following sections (Section 4.3 and 4.5, respectively).

For some of the datasets we experimented with, this simple approach failed even to achieve speedup; on the contrary, the system suffered from performance degradation due to workload imbalance (see). To resolve this problem, we introduce the system-wide BSF technique, in Section 4.4.

4.2 Dataset Shuffling

To explore how to achieve workload balance, our next step was to apply a shuffling technique on the initial dataset (see Figure 6) and then again partition it into N disjoint equally sized chunks. By doing this, the resulting chunk (of the shuffled dataset) that is assigned to each node, contains randomly chosen data series. Specifically, we have considered the following two techniques to shuffle the initial dataset (Algorithm 1):

a. *(1st technique)* Starting from N empty chunks of data series (one for each node), iterate over the data series of the initial dataset and for each data series, randomly select a set and place the data series in this set; the resulting chunks constitute the chunks of the nodes.

b. *(2nd technique)* Let S be the number of data series in the initial dataset. Iterate over the data series of the initial dataset and for each data series, choose a unique position p, $1 \leq p \leq S$, that has not been selected in previous iterations (lines 2- 6), to place it into the shuffled dataset. Upon completion, partition the shuffled dataset into N chunks.

Algorithm 1: Dataset Shuffle

Input
D: sequence of **S** data series

Output
sequence of **S** data series

1st Technique

Local Variables
D': sequence of data series, initially Ø,
$\mathbf{Ch_j}$, $1 \leq j \leq N$: sets of data series, initially Ø

```
1. for each ds in D:
2.     j := choose a random integer in 1..N
3.     place ds into Ch_j
4. return Ch_1, Ch_2, …, Ch_N
```

2nd Technique

Local Variables
D': sequence of S data series, each of them initially ⊥,
chosenPos : sequence of integer size |S| with initial value 0, p: integer

```
1. for each ds in D:
2.     while true:
3.         p := select a random integer in 1..S
4.         if chosenPos[p] = 0:
5.             chosenPos[p] := 1
6.             break
7.     D'[j] := ds
8. return D'
```

Considering the first shuffling technique, the resulting chunks ($\mathbf{Ch_1, Ch_2, \ldots, Ch_N}$) will indeed be disjoint (as required) but the number of data series they contain may be slightly different. Therefore, we avoid using this technique. Instead, we chose the second one, which better meets our requirements of data series distribution across the nodes, since it additionally produces equally sized chunks. By incorporating shuffling, it is possible to achieve better workload balance, since the data series that are close in distance to a query Q are now uniformly distributed among chunks; so, we expect that with a random dataset the nodes will perform more or less the same pruning and populate their priority queues with a similar number of items, during the query answering phase for Q.

However, there are cases, where shuffling the initial dataset may not improve workload balance. For instance, assume N=2 and consider a query Q that in close distance with only a single data series and assume that it turns out this data series to be located in the first chunk after applying the shuffling technique. Therefore, Q is in close distance with only some data series of the first chunk, and it is not in close distance with any of the data series of the second chunk. When n_1 computes the initial BSF, it will find the data series that is in close distance to Q, thus resulting in pruning most of its index tree. In contrast, n_2 will end up to some (kind of) arbitrary data series when calculating the initial BSF, and this may result in examining a lot more index tree paths because its initial BSF may turn

out to be much further from the final answer for Q. In this case, n_2 will possibly populate its priority queues with a lot of elements. This results in a workload imbalance between n_1 and n_2. Moreover, this extra work performed by n_2 is unnecessary. If no care is taken, the performance of DRESS will be greatly affected since n_2 delays the coordinating node from reporting the final answer for Q.

This is the reason why, although we managed to alleviate the system's performance degradation that we faced on some of the datasets we experimented with by applying the shuffling technique, DRESS still achieved no speedup (over the single node MESSI), in some cases (see -c). Moreover, in the discussion up to now, we assumed that it is possible to request from a *computing node* to shuffle the initial dataset and re-distribute it (or the appropriate parts of it) to the *working nodes*, i.e., the ones running DRESS. However, the functionality provided by this computing node, may not be applicable in a real-time environment where response time is crucial since it requires significant computing time to shuffle the dataset. In the following sections, we introduce techniques to resolve these issues.

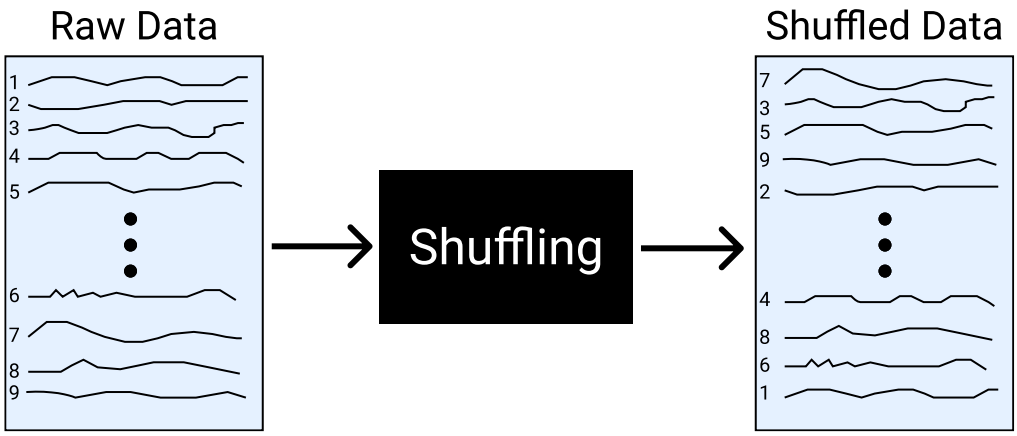


Figure 6 – Split dataset, after shuffling

4.3 System-wide BSF

Up to this point, the techniques we employed (for partitioning and shuffling the dataset) were ignorant to the MESSI algorithm and they could be applied on any distributed algorithm that answers exact similarity search queries to achieve better workload balance. However, as already discussed, are not sufficient to avoid the workload imbalance problem in all cases. This is a fundamental problem to solve in order to achieve good performance and speedup. To discover the cause of this workload imbalance that was responsible for the observed performance degradation, we studied in detail the MESSI algorithm and the single-node techniques it employs. This study resulted in some new multi-node techniques to achieve better workload balance. The first of them is the *emulation of a system-wide best-so-far (SW-BSF)*, which is presented in Algorithm 2 and described below.

Recall that during the query answering phase of every instance of MESSI (where each such instance is executed by a distinct node of DRESS), a single thread computes the initial BSF (lines 2-3); we call this the *coordinator thread* of the node. Then, the coordinator thread initiates ***M*** *worker threads* that work in parallel to prune the tree index based on this initial BSF and to populate the priority queues (line 11). Later, the worker threads calculate (in parallel) the real distances of the corresponding query from each of the data series stored into priority queues. Whenever a worker thread calculates a distance that is better than the currently known BSF, it informs the other threads by atomically updating BSF with the new distance (lines 12-17). By doing so, threads share the best-known BSF value and use it to achieve better pruning.

Additionally, up to now, the nodes of DRESS work independently without sharing the knowledge of the BSF values that they discover. As we saw, this lack of communication may impact performance: a node may unnecessarily calculate real distances with data series having a lower bound distance that is larger than the BSF discovered by some other node. To overcome this problem, nodes must cooperate to avoid computing real distances that are irrelevant based on information only available in some of the nodes.

Therefore, we extended the single-node technique of sharing the BSF to a multi-node setting, as shown in Algorithm 2. This way, a node does not calculate distances that are larger than some other node's BSF. Specifically, whenever the coordinator thread or a working thread of a specific node updates its local BSF (lines 3 and 23, respectively), the coordinator thread informs the other nodes about the new BSF value, by broadcasting it to them (lines 4 and 6-7). Whenever the coordinator thread of a node receives a BSF value (from some other node), it first checks if the received BSF value is larger than its local BSF value, and if this is the case, it updates its local BSF (lines 8-10), thus informing the worker thread of its node about the new value. When a node completes its query answering phase, it sends its partial answer to the coordinator node (lines 11-14), which keeps the minimum of the partial answer and returns it as result.

The above technique has as a result, each node to update its local BSF using the best value among the BSF value discovered by all other nodes. This emulates, in a distributed fashion, the way that the worker threads communicate to each other the newest BSF value in the original single-node MESSI. Using this solution, DRESS achieves a much better workload balance and thus, experiences speedup (over the single-node MESSI) for all the datasets we experimented with. Specifically, using the *SW-BSF*, DRESS achieves approximately 1.7 speedup on the Seismic dataset.

Algorithm 2[4]: System-wide BSF, code for distributed node n_k, $1 \leq k \leq N$

Input
tree index **TreeIndex**, query **Q**

Shared
real number **BSF, workersComplete** initially 0

Code for coordinator thread $\mathbf{t_0}$

```
1.  upon receiving no message, execute once for Q:
2.      Q' := calculate isax summary of Q
3.      BSF := approximate search of Q in TreeIndex using Q'
4.      broadcast message <type: SW_BSF, payload: BSF>
5.      initialize and execute M worker threads with parameters   query Q:=Q and Q':=Q'

6.  upon receiving no message, if BSF has been updated:
7.      broadcast message <type: SW_BSF, payload: BSF>

8.  upon receiving message <type: SW_BSF, payload: newBSF>:
9.      if newBSF < BSF:
10.         atomically update BSF with nodeBSF

11. upon receiving no message:
12.     if workersComplete = M:
13.         send message <type: PARTIAL_ANSWER, payload: BSF, dsAnswer> to n_0
14.         terminate t_0
```

Code for worker thread $\mathbf{t_i}$, $1 \leq i \leq \mathbf{M}$

Input
query data series **Q**, iSAX summary Q'

```
15. prune TreeIndex with Q' and populate priority queues in a round robin fashion
16. q := select the (i % |priority queues|)-th priority queue
17. while true:
18.     while l := dequeue a leaf node from q and l ≠ ⊥
19.         if l.lower_bound_distance_estimation < BSF:
20.             for each data series ds in l with lower bound distance < BSF:
21.                 rd := calculate the real distance of ds from Q
22.                 if rd < BSF:
23.                     atomically update BSF with rd
24.     q := next not finished queue
25.     if q = ⊥:
26.         break
27. atomically increment workersComplete by 1
28. terminate t_i
```

However, there are cases where the BSF may not work well enough. For instance, considering again N=2 and a query Q, assume that all the data series of the first chunk are in close distance to Q, while the data series of the second chunk is not in close distance to Q. In this case, during the query

[4] Algorithm 2 highlights only the points of MESSI where our technique applies and does not provide the full code of MESSI.

answering phase, the BSF calculated by n_1 is always better than the BSF calculated by n_2. So, after n_1 broadcasts its updated values of BSF, n_2 prunes most of its tree index and thus, its priority queues will be populated with a much smaller number of leaf nodes, in comparison to the priority queues of n_1. Additionally, any BSF value broadcasted by n_2 has no impact on the local BSF of n_1 and thus, it does not affect the workload of n_1. Therefore, n_2 may complete its work much faster than n_1 and remain idle for a long period (waiting n_1 to complete its work). This results in workload imbalance. In the following section, we introduce *multi-node work-stealing techniques* to resolve this issue, where (considering the previous scenario) n_2 undertakes part of n_1's work to avoid remain idle and allow n_1 to complete faster.

4.4 Work Stealing

To achieve better workload balance in scenarios where several nodes remain idle waiting for other nodes to complete their calculations, work-stealing can be applied. The basic idea is to let an idle node continue, by acquiring part of others' work. This way it helps others to finish their calculations faster instead of remaining idle. This results in better work balance and thus in better performance.

We introduce two *multi-node work-stealing techniques* for DRESS. By studying the techniques that DRESS inherits from MESSI, we discover candidate parallel computation-intensive mechanisms in MESSI for which applying a work stealing-like technique could be beneficial. Specifically, the first work-stealing technique we implemented extends the priority queues mechanism of MESSI to the multi-node setting of DRESS, without modifying them, whereas the second technique we introduce extends and modifies the mechanisms incorporated by MESSI to prune its tree index and populate the priority queues. These work-stealing techniques can be combined with the *SW-BSF* technique. The combined solution enjoys the benefits of both techniques and had good performance in all scenarios we found problematic.

4.4.1 Stealing work from the Priority Queues

The priority queues that are employed by MESSI, are used to reduce the set of data series for which the real distances from the corresponding query must be calculated. Our first multi-node work-stealing technique extends this mechanism. Briefly, whenever a node finishes its query answering phase for some query, it identifies another node that is still working and helps it by "stealing" work from it. Specifically, the idle node broadcasts a message requesting some work. Busy nodes remove some leaf nodes from their priority queues and send them to the helper node, which is now responsible to process them. When the helper node completes its processing, it initiates a new stealing attempt .

The details are shown in Algorithm 3. When a node completes its local computation, it broadcasts an *availability message* informing other nodes of its *availability* and waits for job offers from them (lines 13-16). Each node that receives an availability message and it is currently in the phase of processing its priority queues (i.e., it is a busy node), replies with a *job offer message* containing the job's description (lines 17-25). Specifically, it contains one or several leaf nodes dequeued from the

corresponding sender node's priority queues. When a node receives a job offer message, it calculates the distances of the data series described by the message with the corresponding query (lines 26-30). After the node processed all the job offer messages it received, it sends another availability message (lines 13-16). In this way, this node extends its own phase of priority queues processing, by stealing items from other nodes' priority queues.

Notice that the nodes that have not yet sent an available message are busy. When a node realizes that all other nodes are available (i.e., have completed processing their priority queues, but may still work on job offers), it sends the partial answer it discovered to the coordinating node (lines 19-22); thus, completes both the answering phase and the work-stealing phase for the corresponding query. The rest of the DRESS algorithm remains the same. Specifically, Algorithm 3 has, as input parameter, a (boolean) flag, called **systemWideBSF,** that identifies whether the *SW-BSF* technique should be applied as well; the blue code of Algorithm 3 implements this functionality.

We now focus on how a job offer describes the work to be done. If the dataset is replicated to all nodes, then this description could be as simple as providing the offsets in the dataset of the corresponding data series. Otherwise, in case only a part of the dataset is available to each node (and these parts are disjoint), then a job offer message should also contain the raw data series. This significantly increases the communication cost for sending the job offer (in comparison to the previous case). Additionally, in this case, following the above communication protocol could result in receiving multiple (large) job offer messages in some available node, thus the system in total pays a big communication overhead without the corresponding performance benefits being guaranteed. This is so, since the available node may not be able to process all of them before the senders complete their (remaining) calculations. Therefore, in this case, the above communication protocol can be modified so that an available node does not receive job offer messages from all other busy nodes, but only from a subset of them that it will choose. To so do, the available node can randomly choose one or a set of the busy nodes and send an availability message only to them.

Algorithm 3[5]: Work-Stealing on Priority Queues, code for distributed node n_k, $1 \leq k \leq N$

Input
tree index **treeIndex**, query **Q**, boolean **systemWideBSF**

Shared
real number **BSF**, data series **dsAnswer**, integer **workersComplete** initially 0,
integer **workSteal** initially 0, boolean **terminate** initially **false**,

Code for coordinator thread t_0

Local Variables
set of ids **availableNodes** initially ∅, boolean **availabilitySent** initially false

1. **upon receiving no message**, execute once for **Q**:
2. Q' := calculate isax summary of **Q**
3. **BSF** := approximate search of **Q** in **TreeIndex** using Q'
4. if **systemWideBSF** = true:
5. **broadcast message** <type: SW_BSF, payload: BSF>
6. initialize and execute M worker threads with parameters Q:=**Q** and Q':= Q'

7. **upon receiving no message**, **if BSF** has been updated:
8. if **systemWideBSF** = true:
9. **broadcast message** <type: SW_BSF, payload: **BSF**>

10. **upon receiving message** <type: SW_BSF, payload: newBSF>**:**
11. **if** newBSF < **BSF**:
12. atomically update **BSF** with nodeBSF

13. **upon receiving no message:**
14. if **workersComplete** = M and **workSteal** = 0 and **availabilitySent** = false:
15. **broadcast message** <type: AVAILABLE, payload: **k**>
16. **availabilitySent** := true

17. **upon receiving message** <type: AVAILABLE, payload: nodeId>**:**
18. add nodeId to **availableNodes**
19. **if** |**availableNodes**| **= N -1:**
20. **terminate** := **true**
21. **send message** <type: PARTIAL_ANSWER, payload: BSF, dsAnswer> **to** n_0
22. terminate t_0
23. **if** priority queues **not** empty:
24. leafNodes := dequeue leaf nodes from priotiry queues in a round robin fashion
25. **send message** <type: JOB_OFFER, payload: leafNodes> **to** n_{nodeId}

26. **upon receiving message** <type: JOB_OFFER, payload: leafNodes>**:**
27. **for each** leaf node in leafNodes:
28. add leaf node to priority queues in a round-robin fashion
29. atomically add |leafNodes| on **workSteal**
30. **availabilitySent** := false

[5] Algorithm 3 highlights only the points of MESSI where our technique applies and does not provide the full code of MESSI.

Code for worker thread t_i, $1 \leq i \leq M$

Input

query data series **Q,** iSAX summary Q'

```
31.  prune TreeIndex with Q' and populate priority queues in a round-robin fashion
32.  q := select the (i % |priority queues|)-th priority queue
33.  while terminate = false:
34.      while true:
35.          while l := dequeue a leaf node from q and l ≠ ⊥
36.              if l.lower_bound_distance_estimation < BSF:
37.                  for each data series ds in l with lower bound distance < BSF:
38.                      rd := calculate the real distance of ds from Q
39.                      if rd < BSF:
40.                          atomically update BSF with rd and dsAnswer with ds
41.              if not 1st iteration of the while of line 34:
42.                  atomically subtract 1 from workSteal
43.          q := next not finished queue
44.          if q = ⊥:
45.              break
46.      if 1st iteration of the while of line 33:
47.          atomically increment workersComplete by 1
48.  terminate ti
```

4.4.2 Stealing work from Tree Index

The MESSI technique used to prune the tree index is also computation-intensive and thus is a second candidate for applying work-stealing. During query answering, MESSI first computes the initial BSF. Then, based on it, it prunes the tree index, so that it reduces the number of lower bound distance calculations and of real distance calculations. The tree index leaves that cannot be pruned are inserted into the priority queues. Our second multi-node work-stealing technique extends this mechanism so that an available node can acquire and prune parts of the tree index of some other busy node. Additionally, the available node processes the non-pruned tree index leaves of these parts, using its priority queues. To do so, the original MESSI technique must be modified, as described below.

The pruning of the tree index of each node is divided into parts. Then, each node starts by pruning some part of the tree index (instead of pruning the whole tree index at once), starting from the first part, and populates the priority queues accordingly. Then, it processes the leaf nodes it inserted into the priority queues. As soon as it completes, it continues with the next part of the tree index. By doing this, an available node (i.e., a node that finished its query answering phase for some query) can now acquire a part of the tree index of some busy node (i.e., a node that is still working) and process it (i.e., prune it, populate priority queues, and process them). This way, an available node can acquire part of the work of some other busy node. Specifically, the busy node informs the helper node about the part of the index tree that it can process. When the helper node completes this job, it continues by stealing more work from this or some other busy node.

Notice that the modification of the MESSI algorithm described above so that it works gradually on parts of the tree index (as opposed to working on the complete tree index at once) is necessary since

otherwise, any part of the tree index acquired to be processed by some available node, it would also be processed by the busy node. In this case, no performance benefit is gained since parts of the tree are simply processed by more than one node; this could also result in performance degradation.

In more detail, as Algorithm 4 shows, the tree index is divided into parts with each one containing a specific number of subtrees, e.g., 10000 subtrees; recall that the tree index consists of up to 2^{16} subtrees. Starting with the normal execution, during the query answering phase of DRESS, each node first computes the initial BSF by accessing its tree index (lines 1-3), similarly to MESSI. Subsequently, rather than pruning the whole tree index (like in MESSI), in DRESS the node starts by processing only a part of it (lines 16-20). This processing includes pruning this part of the tree index, appropriately populating priority queues, and processing them (lines 37-49). Only then, a node continues by processing the next unprocessed part of the tree index (lines 17).

Then, similarly to our first work-stealing technique, when a node completes its local calculations, it broadcasts an *availability message* (lines 23-24) informing other nodes of its *availability* and waits for job offers from them. Each node that receives an availability message and it is still busy, replies with a *job offer message* containing the part of the tree index to be processed (lines 27-33). When a node receives a job offer message from some busy node (lines 34-36), it uses the tree index of this busy node's chunk of the dataset to process the part of the tree index described by this message. After the node processed all the job offer messages it received, it broadcast another availability message and waits for new job offers. This way, this node extends its own phase of tree index processing, by stealing items from other nodes' tree indexes. When a node realizes that all other nodes are available (i.e., have completed processing their tree indexes, but may still work on job offers), it sends the partial answer it discovered to the coordinating node (lines 28-29); thus, completes both the answering phase and the work-stealing phase for the corresponding query. The rest of the DRESS algorithm remains the same. The blue part of Algorithm 4 illustrates again how the *SW-BSF* technique can be also incorporated.

We remark that in order for an idle node to be able to steal jobs from some other busy node, it should have access to the part of the tree index of the busy node's chunk. If this part of the tree index is not yet created, it must be constructed by the idle node that receives this job offer. To alleviate this problem, it might be beneficial for a node to create not only the tree index of its own chunk of data series but also the tree indexes of all other nodes' chunks. This can be done before the node initiates its query answering phase, so that the creation of the tree indices is performed outside the critical path of the query answering phase. Note that the idle node may steal several jobs from the same busy node. Thus, the cost of creating the tree index for the entire busy node's chunk (and not just a part of it), when the idle node receives the first job from this busy node, could be amortized (depending on later job offers). For these reasons, we selected to implement this approach in DRESS.

Algorithm 4 describes only the query answering phase of DRESS. It receives as input parameter the **TreeIndexes** array that contains the (already created) tree indexes of all the chunks of the dataset.

Algorithm 4: Work-Stealing on Tree Index, code for distributed node n_k, $1 \leq k \leq N$

Input
array of tree indexes **TreeIndexes** with N positions, query **Q**, boolean **systemWideBSF**

Shared
real number **BSF**, data series **dsAnswer**, integer **workersComplete** initially 0

Code for coordinator thread t_0

Local Variables
set of ids **availableNodes** initially ∅, boolean **availabilitySent** initially false, boolean **anweringPhaseComplete** initially false, set of <tree index parts, nodeId> pairs **treeIndexParts** initially ∅, boolean **init** initially false

1. **upon receiving no message**, execute once for **Q**:
2. Q' := calculate isax summary of **Q**
3. **BSF** := approximate search of **Q** in **TreeIndexes**[k] using Q'
4. if **systemWideBSF** = true:
5. **broadcast message** <type: SW_BSF, payload: BSF>
6. for each index part ip of **TreeIndexes**[k]:
7. add <ip, n_k> in **treeIndexParts**
8. **init** := true

9. **upon receiving no message**, if **BSF** has been updated:
10. if **systemWideBSF** = true:
11. **broadcast message** <type: SW_BSF, payload: **BSF**>

12. **upon receiving message** <type: SW_BSF, payload: newBSF>**:**
13. if newBSF < **BSF**:
14. atomically update **BSF** with nodeBSF

15. **upon receiving no message** and **init** = true**:**
16. if executed for first time or **workersComplete** = M:
17. <treePart, nodeId> := remove an element of **treeIndexParts**
18. if treePart ≠ ⊥:
19. **workers Complete :**= 0
20. initialize and execute M worker threads with parameters **Q**, Q', treePart,nodeId
21. else:
22. **anweringPhaseComplete** := true
23. if **anweringPhaseComplete**=true and **workersComplete** = M and **availabilitySent** = false:
24. **broadcast message** <type: AVAILABLE, payload: **k**>
25. **availabilitySent** := true
26. **upon receiving message** <type: AVAILABLE, payload: nodeId>**:**
27. add nodeId to **availableNodes**
28. **if** |**availableNodes**| **= N-1:**
29. **send message** <type: PARTIAL_ANSWER, payload: BSF, dsAnswer> **to** n_0
30. terminate t_0
31. <treePart, id> := remove an element of **treeIndexParts**
32. **if** treePart ≠ ⊥ and id = k:
33. **send message** <type: JOB_OFFER, payload: treePart> **to** n_{nodeId}

34. **upon receiving message** <type: JOB_OFFER, payload: treePart> from n_{nodeId}:
35. add <treePart, nodeId> in **treeIndexParts**
36. **availabilitySent** := false

Code for worker thread t_i, $1 \leq i \leq M$,

Input
query data series **Q**, iSAX summary Q', tree index part treePart, integer nodeId

```
37.  prune treePart of TreeIndexes[nodeId] with Q' and populate priority queues in a round robin fashion
38.  q := select the (i % |priority queues|)-th priority queue
39.  while true:
40.      while l := dequeue a leaf node from q and l ≠ ⊥
41.          if l.lower_bound_distance_estimation < BSF:
42.              for each data series ds in l with lower bound distance < BSF:
43.                  rd := calculate the real distance of ds from Q
44.                  if rd < BSF:
45.                      atomically update BSF with rd and dsAnswer with ds
46.      q := next not finished queue
47.      if q = ⊥:
48.          break
49.  atomically increment workersComplete by 1
```

Additionally, to be able to apply this work-stealing technique, the helper node must have access to the whole dataset; thus, the dataset must be replicated to all nodes. To alleviate the problem of increased space consumption that this approach incurs, a similar approach could be to replicate in each node only some of the other chunks, in which case this node could only (create the tree indexes and thus) steal work from the corresponding nodes.

4.5 Parallel Execution of Multiple Queries

In this paper, we mainly focus on techniques that speedup the execution of a single query, thus assuming that the queries are not independent to one another, i.e., which the next query will be may depend on the answer of previous queries. In a system with independent queries that are provided in batches, additional parallelization techniques can be employed. For instance, instead of having all the nodes answering a single query, the set of the queries can be partitioned into subsets with each subset assigned to each node. This way each query is answered by a single node. Thus, during query answering, each node undertakes a number of queries to answer. We remark that to implement this simple approach, each node should have locally replicated the whole dataset. However, this is not always possible, since nowadays the size of a dataset can reach up to several terabytes.

Although designing and experimenting with parallelization techniques for batches of queries is out of the scope of this paper (where we focus on parallelizing the answering procedure of a single query), we have implemented the above simple approach. Our experiments (Section 6.3.4) show that a simple implementation of the above technique (in which no other of the DRESS techniques described in previous sections, have been applied) exhibits speedup of 1.8. This is an indication that parallelizing the execution of multiple queries is a promising approach for further future work.

5. DRESS Algorithm

Briefly, each node of DRESS executes an enhanced instance of the MESSI algorithm, where i) a single node undertakes the task of partitioning (if needed) and distributing the dataset to the nodes and ii) the nodes are coordinated so that they initiate their query answering phase only after all of them completed their tree index construction phase. Then, in case DRESS works with dependent queries, it additionally coordinates the nodes so that they iii) process each query one after the other, i.e., the query answering phase of the next query can start only after the completion of the query answering phase of the previous query, and for each query q, iv) report the result (for q) on the part of the dataset that each one of them works, and v) select the best result among them to be the answer of q. Otherwise, in case DRESS works with a batch of independent queries, each node undertakes the task of answering a subset this batch, without coordinating with other nodes.

In this section, we present the complete pseudocode of the DRESS algorithm.

5.1 Pseudocode

Algorithm 5 and Algorithm 6 present the pseudocode of DRESS. Recall that in Sections 4.3 to 4.4, we have already described the communication protocol that each of our multi-node techniques incorporates during the query answering phase. To avoid repetition, we incorporate the pseudocode of these techniques by referencing to Algorithms 1-4, when required. We do the same also for the shuffling techniques presented in Section 4.2. We assume that the distributed system consists of one *coordinator node* (n_0) and N *worker nodes* (n_1, n_2, …, n_N). Algorithm 5 presents the pseudocode of DRESS for the coordinator node and Algorithm 6 presents the pseudocode of DRESS for the worker nodes. We remark that the coordinator node is comprised of only a single thread, while each of the worker nodes uses all its available threads, as explained in the following.

The coordinator node receives as input parameters the dataset (**Dataset**), a set containing the queries (**Q**), and five boolean variables indicating whether the shuffling of the dataset (**shuffling**), the parallel execution of queries (**parallelQueries**), the work-stealing from the priority queues (**wStealing1**), the work-stealing from the tree index (**wStealing2**), and the system-wide BSF (**systemWideBSF**) techniques will be used. We remark that DRESS supports all the meaningful combinations of our multi-node query answering techniques, which are: i) parallel execution of queries, ii) system-wide BSF (SW-BSF), iii) work-stealing from the queues together with (or without) SW-BSF, and iv) work-stealing from the tree index with (or without) SW-BSF.

5.1.1 Coordinator node

We start by describing the pseudocode of DRESS for the coordinator node (Algorithm 5). When **shuffling** is true, DRESS shuffles the dataset (lines 2-3), using the second approach of Algorithm 1. In any case, the coordinator node should start by sending to each worker node a message with type INIT (lines 4-9), containing i) the appropriate set of data series that the worker node needs to process and ii) the flags that determine which of the multi-node query answering techniques (i.e., *SW-BSF*, work stealing, and parallel execution of queries) should be used. Specifically, in case either **parallelQueries**

or **wStealing2** is true, then the former is the whole dataset (lines 4-5), as required by the corresponding techniques. On the other hand, in case either **systemWideBSF** or **wStealing1** is true (line 6), the coordinator node first partitions the dataset into chunks and then sends only one of them to each worker node (lines 7-9); so, the aforementioned appropriate set is this chunk. In both cases, the coordinator node sends the appropriate set of data series to each of the worker nodes (lines 5 and 9, respectively).

When a worker node receives the INIT message from the coordinator node, it creates its tree index and after it completes, replies with a completion message of type INDEX_COMPLETE. Then, upon receiving an INDEX_COMPLETE message, the coordinator node increments a local variable (**indexComplete**) maintaining the number of the worker nodes that completed their tree index construction phase. When the coordinator node received messages from all the worker nodes that have finished with the tree index construction (line 12), i.e., **indexComplete**=N, it proceeds and sends the queries that needs to be answered (lines 13-19). The number of the queries it sends, depends on whether the queries are independent or not. In the former case (line 13), it splits the batch of the queries to equal subsets and send each subset to a worker node to answer them in parallel (lines 13-16). Recall that in this case, each node has already received and created a tree index for the complete dataset, as required. In the latter case, it broadcasts only a single query to all worker nodes (lines 18-19) to coordinate the answering of queries one after the other, as explained below.

Whenever a worker node n_j, $1 \leq j \leq N$, completes its answering phase for some query q, it sends a completion message to the coordinator node containing the answer it calculated. The coordinator node maintains a local set **Answers** to store the answers it receives. In case independent queries are used, the message sent by n_j is of type ANSWER and contains the final answer of q (recall that in this case each query is only processed by a single worker node). Then, upon receiving this message, the coordinator node adds in **Answers** the final answer it contains together with q (lines 21-22).

On the other hand, if dependent queries are used, the message sent by n_j is of type PARTIAL_ANSWER and contains a partial answer for q. Recall that in this case each query is processed by all the worker nodes. So, the coordinator node must determine the best answer for q and keep track of the number of worker nodes that reported a partial answer for q. To implement these, the **Answers** set also maintains the distance of each partial answer from q (which is sent by n_j together with its partial answer, to avoid re-calculating it) and the number (**nodes**) of completion messages received for q, respectively. When the coordinator receives a partial answer for q, it first checks whether it already maintains a record for q in **Answers** (lines 24-25 and 27). If no such record exists, i.e., this is the first partial answer the coordinator node receives for q, it is added in **Answers** together with its distance and the value 1 for **nodes** (lines 24-26). Otherwise, if this record exists and this partial answer is more relevant from the stored one (i.e., it has a smaller dinstace from q), the record is updated (lines 28-29) so that it stores the newly received partial answer and its distance from q. In both cases, the **nodes** number of this record is incremented by one (indicating that another worker node has completed with the answering of q). As soon as the coordinator node receives N partial

answers (line 31), it follows that this record contains the final answer of q. So, it broadcasts the next query that needs to be answered (lines 32-33). This way DRESS coordinates the working nodes to answer queries one after the other.

Algorithm 5: DRESS, code for coordinator node n_0

Input
set of data series **Dataset**, set of (query) data series **Q**, boolean **shuffling,** boolean **parallelQueries**, boolean **wStealing1**, boolean **wStealing2**, boolean **systemWideBSF**

Local Variables:
set of <data series **query**, data series **answer**, real number **distance,** integer **nodes**> tuples **Answers** initially ∅, integer **indexComplete** initially 0

```
1.  upon receiving no message, execute once:
2.      if shuffling = true:
3.          Dataset := execute Algorithm 1 (second technique) with parameter D := Dataset
4.      if parallelQueries = true or wStealing2 = true:
5.          broadcast message <type: INIT, payload: Dataset, parallelQueries, false, wStealing2, ystemWideBSF>
6.      else if systemWideBSF = true or wStealing1 = true:
7.          for each node n_k, 1 ≤ k ≤ N:
8.              chunk := the k-th part of Dataset                    // each part has size |Dataset|/N
9.              send <type: INIT, payload: chunk, false, wStealing1, false, systemWideBSF> to n_k

10. upon receiving message <type: INDEX_COMPLETE, payload: ⊥>:
11.     indexComplete := indexComplete + 1
12.     if indexComplete = N:
13.         if parallelQueries = true:
14.             for each node n_k, 1 ≤ k ≤ N:
15.                 queries := the k-th part of Q                    // each part has size |Q|/N
16.                 send message <type: QUERIES, payload: queries> to n_k
17.         else
18.             select any data series q of Q
19.             broadcast message <type: QUERY, payload:q>

20. upon receiving message <type: ANSWER or PARTIAL_ANSWER, payload: Q, answer, distance>:
21.     if parallelQueries = true
22.         add <Q, answer, ⊥, ⊥> in Answers
23.     else:
24.         tup := get the tuple of Answers where q = Q
25.         if tup = ⊥:
26.             add <Q, answer, distance, 1> to Answers
27.         else:
28.             if tup.distance < distance:
29.                 <tup.answer, tup.distance> := <answer, distance>
30.             tup.nodes := tup.nodes +1
31.         if tup.nodes = N and |Answers| < |Q|:
32.             select a data series q of Q for which no tuple t with t.query = q exists in Answers
33.             broadcast message <type: QUERY, payload:q>
34.     if |Answers| = |Q|:
35.         broadcast message <type: TERMINATE, payload: ⊥>
36.         for each tuple t in Answers:
37.             report t.answer as the data series answer to query t.query
38.         terminate
```

Independently of whether the queries are independent or dependent, when all the queries are answered (line 34), the coordinator node broadcasts a termination message to the worker nodes (line 35), reports the discovered answers (lines 36-37), and terminates itself (line 38).

Algorithm 6: DRESS, code for worker node n_k, $1 \leq k \leq N$

Local Variables: set of data series **Dataset**, boolean **parallelQueries**, boolean **wStealing1**, boolean **wStealing2**, boolean **systemWideBsf**, array of N slots **IS** with each slot containing a set of iSAX summaries that is initially Ø, array of N slots **Index** with each slot containing a tree index initially ⊥

Code for coordinator thread t_0

39. upon receiving message <type: INIT, payload: dataset, independent, s1, s2, bsf>:
40. <**Dataset**, **parallelQueries** > := <dataset, independent>
41. <**wStealing1**, **wStealing2**, **systemWideBsf**> := <s1, s2, bsf>

// calculate iSAX Summaries

42. if **parallelQueries** = true or **wStealing1** = true or (**wStealing2** = false and **systemWideBsf** = true):
43. **IS**[k] := calculate the iSAX summaries of the data series in **Dataset** using M threads *(in a way similar to MESSI)*
44. else if **wStealing2** = true:
45. for each m, $1 \leq m \leq N$:
46. **IS**[m] := calculate the iSAX summaries of the data series in the m-th part of **Dataset** using M threads *(in a way similar to MESSI)*

// create Index(es)

47. for each m, $1 \leq m \leq N$, with **IS**[m] $\neq$ Ø:
48. **Index**[m] := create a tree index for the elements of **IS**[m] using M worker threads *(in a way similar to MESSI)*
49. send message <INDEX_COMPLETE, payload: ⊥> to n_0

50. upon receiving message <type: QUERIES, payload: queries>:
51. for each q ∈ queries:
52. <answer, distance> := find the exact answer to query q using **Index**[m] and M worker threads *(in a way similar to MESSI)*
53. send message <ANSWER, payload: q, answer, ⊥ > to n_0

54. upon receiving message <type: QUERY, payload: query>:
55. if **wStealing1** = true:
56. execute Algorithm 3 with parameters **TreeIndex** := **Index**[k], **Q** := query, **systemWideBsf**
57. else if **wStealing2** = true:
58. execute Algorithm 4 with parameters **TreeIndexes** := **Index**, **Q** := query, **systemWideBsf**
59. else: *(systemWideBSF = true)*
60. execute Algorithm 2 with parameters **TreeIndex** := **Index**[k] and **Q** := query

61. upon receiving message <type: TERMINATE, payload: ⊥>:
62. terminate

5.1.2 Worker nodes

We continue with the description of the pseudocode of DRESS for a worker node n_j, $1 \leq j \leq N$ (Algorithm 6). Each worker node has a single coordinator thread t_0 that is responsible to communicate with the (coordinator threads of the) other nodes. Whenever required, n_j initializes M worker threads to execute the various phases of MESSI, i.e., iSAX summarization, index construction, and query answering. Initially, the coordinator thread waits until it receives an INIT message from the coordinator node, which contains the dataset (**Dataset**) that the node must process (either the complete dataset or a chunk of it, as described in Section 5.1.1) and the flags that determine which of the multi-node query answering techniques (i.e., *SW-BSF*, work stealing, and parallel execution of queries) should be used (lines 40-41).

Specifically, if the **parallelQueries** is true, or the **wStealing1** is true, or the **wStealing1** is false and the **systemWideBSF** is true (line 42), then n_j must build a single tree index for the received dataset (**Dataset**). So, in this case, it continues by calculating the iSAX summaries of the data series contained in **Dataset** (line 43).

On the other hand, if the **wStealing2** is true the n_j must create a tree index on every chunk of the dataset to be able later on, to apply work stealing from the other nodes tree indexes; thus, it calculates the iSAX summaries for every chunk of the received dataset (lines 44-46). In any case, each iSAX summarization (on each chunk or in the whole dataset) is calculated as MESSI does, from M worker threads in parallel.

When the worker threads finish with the iSAX summarization, continues with the building of the tree(s) index (lines 47-48), depends on who many tree indexes must construct as we discuss above. When n_j finishes with the tree index(es) construction, it sends an INDEX_COMPLETE message to the coordinator node (line 49). Thus, the coordinator node can send the queries that need to be answered. If the coordinator node, sends to n_j independent with each other queries with the QUERIES message, the worker nodes just answer the queries one after the other (lines 51-52), as MESSI does, and for each query sends the final answer to the coordinator node (line 53). On the contrary, if the coordinator node sends a query with the QUERY message, then the n_j, runs the appropriate algorithm from section 4, based on the received flags (lines 40-41) . More specifically, if the **wStealing1** is true n_j runs the Algorithm 3, work-stealing on the priority queues (lines 55-56), else, if the **wStealing2** is true it runs the Algorithm 4, work-stealing on the tree index (lines 57-58), else, if only the **systemWideBsf** is true it runs the Algorithm 2, emulating the *SW-BSF* (lines 59-60). One sidenote, Algorithm 3 and Algorithm 4 will incorporate the *SW-BSF*, if the **systemWideBsf** is true. When n_j has answer a query it sends the partial answer to the coordinator node. Finally, when the coordinator node has collected the partial answer for all the queries broadcast a termination message to the worker nodes. As soon as n_j receives a TERMINATE message from the coordinator node, it terminates (lines 61-62).

5.2 Implementation Details

In this section, we present algorithmic changes (Section 5.2.1) of the DRESS algorithm (presented in Section 6) and some details about the implementation of DRESS (Section 5.2.2) that we used for our evaluation; our evaluation is later described in Section 6.

5.2.1 Algorithm modifications

In this section, we describe some modifications we made on the pseudocode of DRESS (Algorithm 5 and Algorithm 6) when implementing it. We start by describing the modifications regarding the coordinator node and the coordinator thread (of each worker node).

1. The coordinator node does not send the data series that each worker node must process (lines 5 and 9, Algorithm 5), but the system is initialized either with the complete dataset (if it fits) or with its corresponding chunk residing in the local disk of each worker node. So, in the former case, each of the worker nodes determine (based on its id) the chunk of the dataset it should work on and load it in memory; while, in the latter case each worker node simply loads its chunk in memory.
2. When each worker node completes its index construction phase, it does not send INDEX_COMPLETE messages to the coordinator node (line 10, Algorithm 5 and line 11, Algorithm 6). Instead, a *distributed index construction phase completion barrier* is used to synchronize the worker nodes, so that only after all of them reach this barrier, they start to answer the queries.
3. The coordinator does not send each query to the worker nodes (lines 10-19, Algorithm 5), but all the queries already reside in each worker node's disk and are loaded in memory during DRESS initialization.
4. Since the coordinator does not send queries to the worker nodes, to be able to simulate dependent queries, i.e., the worker nodes collaborate to answer a single query at each point in time, we add a *distributed query answer completing barrier*, so that only after all of them reach this barrier, they continue by answering the next query.
5. By applying the above modifications on DRESS, we have removed a lot of the work of the coordinator node, which is now responsible only for collecting the (partial) query answers and reporting them. Therefore, in our DRESS implementation, the role of the coordinator node is assigned to one of the worker nodes, which after it completes its query answering phase, simply waits answers of other worker nodes.
6. Similarly, regarding a single worker node, its coordinator thread is one of the worker threads (and not a separate one, as described in Algorithm 6). So, additionally to performing its calculations, this thread is periodically communicating, in a non-blocking, with the other worker nodes to implement DRESS multi-node techniques.

We now describe some modifications we applied on the work-stealing technique from the tree index. First, we create all the required tree indices outside the critical path of the query answering phase. So, since (as described later in Section 6) we experimented with only two worker nodes for the evaluation of DRESS, each worker node initially creates the tree index of its own chunk and then continues by creating the tree index for the chunk of the other worker node. Our second modification regards the reduction of the communication latency of this work-stealing technique. Specifically, after completing its query answering phase, instead of requesting and waiting to receive job offer(s) from the busy node, the idle node immediately sends a message with the part of work that undertakes and then processes parts of the busy node's tree index, starting from the last one, i.e., in the opposite order, since the busy node processes parts staring from the first one. When the busy node receives that message, it reduces the number of the subtrees that needs further processing. This process continues up until the idle and busy worker nodes reach some busy node's tree index part that is already processed.

5.2.2 Message-Passing Interface (MPI)

We designed DRESS for a distributed memory system that consists of a collection of multi-core machines, called *nodes*. These nodes are connected using a local ethernet network and communicate through the Message-Passing Interface (MPI) [25]. Although, a program running on a single machine is called *MPI process*, for simplicity, we will refer to it as a *node*.

Every node starts by calling the MPI_Init function that appropriately allocates the required resources to initialize the MPI system. Additionally, upon completing each node calls the MPI_Finalize function so that MPI deallocates its resources. To implement the *distributed index construction phase completion barrier* and the *distributed query answer completing barrier*, that we discussed in Section 5.2.1, we used the MPI_Barrier function that offers a distributed synchronization point among the nodes.

MPI allows nodes to communicate with each other using either *point-to-point* or *collective* communications. In the former case, two nodes communicate by sending direct messages to each other. Specifically, the sender calls an *MPI send function* and the receiver calls the corresponding MPI *receive function*. In the latter case, a collective communication involves several processes and all of them are calling the same *MPI collective function*, e.g., for broadcasting a message. In both cases, the nodes are obliged to communicate over an *MPI communicator*, which contains the collection of processes that can communicate with each other. MPI offers a default communicator called MPI_COMM_WORLD, which includes all the nodes of the system. Additionally, MPI allows user-defined communicators, where a programmer can designate a (sub)set of the system's nodes that are able to communicate with each other (using this user-defined communicator).

Recall that the nodes of DRESS communicate during their query answering phase to implement the work-stealing and the SW-BSF multi-node techniques. To avoid performance degradation, the *non-blocking* MPI communication functions are used so that i) a sender node continues its execution

before its message reaches the receiver(s) (as is the case with the blocking communication) and ii) a receiver node can test whether a message has arrived (and avoid blocking until it arrives). More specifically, to implement the work-stealing technique, whenever the idle node needs to send a message to the busy node, it calls the non-blocking MPI_Isend function. Then, to receive a message, the busy node periodically calls the non-blocking MPI_Irecv function to test whether a message has arrived (and receive it if this is so).

Moreover, to implement the SW-BSF technique, whenever a node needs to send a new BSF value to all other worker nodes, it calls the non-blocking MPI_Ibcast function that implements a broadcast collective operation. To distinguish between the (single) sender node and the receiver nodes, MPI_Ibcast takes as a parameter the identifier of the node that initiated the broadcast operation (i.e., the one sending a message). In case this parameter is equal to the current node's identifier, this node is the sender; otherwise, if it is different, the current node is one of the receivers. In order all nodes to be able to broadcast their BSF values, we have created one communicator for each one of them (using the MPI_Comm_dup function) on which this node can be the sender and all the other the receivers. Then, periodically each node tests these communicators, excluding its own, for arrived (broadcast) messages.

6. Evaluation

In this section, we provide the experimental analysis of the DRESS algorithm, and we compare its performance with the performance of MESSI [1].

6.1 Experimental Setting

We conducted our experiments on a distributed system with 2 nodes, namely n_1 and n_2. Specifically, we used a cluster of two nodes, where each node is an Intel Xeon E5-2630 v3 CPUs comprised of 2 sockets with each one having 8 cores. Each core can run two hyperthreaded logical cores, resulting in a total of 32 logical cores per node. Additionally, each node has 256 GB main memory (DRAM) and 500GB secondary storage capacity. The nodes communicate using message passing over a 100Gbps Ethernet network connection. We employ OpenMPI 4.0.3rc4. Our code is written in C and compiled using the mpicc compiler of OpenMPI. Each node runs CentOS Linux, release 7.3.1611.

6.2 Experimental Datasets and Benchmarks

We experimented with both synthetic and real datasets, with each dataset containing 100 million data series. Each data series contains 256 real numbers (of 4 bytes). Thus, the size of each dataset is 100GB. A z-normalization function is applied on the data series of both the datasets and the queries.

Our data sets are similar to those used for MESSI and were provided by the authors of the MESSI paper. We used two datasets. The first dataset is a *randomly generated dataset* produced using a random walk data series generator[6]. This kind of data series generation has been extensively used in the past and it has been shown to model real-world financial data [5], [26]. With the same process, the 100 query data series are generated. The second dataset, called *Seismic*, contains data series that describe seismic signals around the world. Specifically, for creating this dataset, the IRIS Seismic Data Access repository [27] was used to gather 100M data series representing seismic waves from various locations, for a total size of 100GB (since the size of each data series is 1024 bytes). In this case, our experiments use 100 queries that have already been produced by applying a synthetic series generator on the first 100 data series of Seismic, following the same process as previous papers [MESSI, PARIS, PARIS+]. Specifically, a noise that was progressively incremented was applied on each of these data series of Seismic, to produce the corresponding query.

Considering the execution of queries, we follow two approaches. In the first approach, the queries are executed sequentially, one after the other, to simulate a scenario where the external environment (e.g., users) present a new query based on the answer received for some of the previous queries. In this case, each query is answered with the collaboration of both nodes. Using this approach, we provide experiments for a) the dataset shuffling technique, b) the *SW-BSF technique of DRESS*, and c) the

[6] A random real number (of 4 bytes) is repeatedly drawn from a Gaussian distribution N(0,1), with each new number being added to the value of the last one. Then, 256 consecutive such randomly generated real numbers constitute a data series.

work-stealing technique on the tree index. In the second approach, we model a scenario where independent queries can be provided in batches by the external environment. In this case, each node undertakes and answers a part of the batch of the queries, independently from other nodes. We use this approach to test the parallel execution of multiple queries in DRESS (Section 4.5), whenever each node maintains a replica of the whole dataset.

In the figures presented in the rest of this section, we use some abbreviations to refer to our experiments, listed in Table 1, together with a brief description. These experiments are applied in both the random and the Seismic datasets. More specifically, a first set of experiments run MESSI on the whole 100GB dataset (called MESSI) and on each half of it (called MESSI-2nd-half and MESSI-1st-half, respectively). The rest of the experiments run DRESS on the whole 100GB dataset. DRESS runs on the initial dataset (called DRESS) and on the shuffled version of the initial dataset (called DRESS-shuffled), which has been produced using the second shuffling technique presented in Algorithm 1. Additionally, we run DRESS on the initial dataset using multi-node techniques, namely the *SW-BSF* (called DRESS-SW-BSF), the work stealing on tree index (called DRESS-work-steal), and the parallel execution of multiple queries (called DRESS-parallel-queries).

We note that experimenting with the work stealing on priority queues multi-node technique is left for future work.

Name	**Interpretation**
MESSI	on the whole (100GB) dataset
MESSI-1st-half	on the first half (50GB) of the dataset
MESSI-2nd-half	on the second half (50GB) of the dataset
DRESS	each of the two nodes processing a chunk of 50GB
DRESS-shuffled	shuffled dataset
DRESS-SW-BSF	system wide BSF
DRESS-work-steal	work stealing on tree index
DRESS-parallel-queries	multiple queries executed in parallel

Table 1 - Experiments Abbreviations

Each experiment is repeated 10 times and we report the average time elapsed for the metrics we count. The first set of metrics include the time elapsed for completing each of the three phases of DRESS: i) the *summarization time* to calculate the iSAX summaries of the raw data series and store each one of them in the appropriate iSAX buffer, ii) the *tree index construction time* to construct the index tree using these iSAX buffers, and iii) the *query answering time*, i.e. the average time to answer a query (100 queries are executed and averages are calculated). More specifically, in each node, to count the summarization time, the coordinator thread starts the timer just before creating the worker threads and stops it just after all the worker threads reach the tree index construction barrier (which means that the summarization phase of DRESS on this node has completed). At this latter point, the

coordinator thread starts the timer to calculate the tree index construction time and stops it just after all the worker threads reach the query answering barrier (which means that the summarization phase of DRESS on this node has completed). Finally, for each query, the coordinator thread starts a timer to count the time elapsed to answer this query and stops it just after all the worker threads reach the query answering barrier (for this query). Since, DRESS is an in-memory algorithm, we do not measure the time required for loading (into memory) the data series that each worker node need to process.

Additionally, we report a second set of metrics, including the average number of all the *lower bound distances* and all the *real distances*, calculated to prune the tree index and process the priority queues.

6.3 Experimental Analysis

In this section, we present the results of our experimental analysis and argue about the observed behavior. We start by presenting our results using the randomly generated dataset (Section 6.3.1) and we continue with our results using the Seismic dataset (Section 6.3.2 and 6.3.3). Each of the (two) tested datasets is 100GB.

6.3.1 Randomly generated dataset

Figure *7*-a, presents our results for the **iSAX summarization phase** of DRESS and MESSI on the 100GB randomly generated dataset, and MESSI-1st-half and MESSI-2nd-half on each of the two 50GB chunks of the 100GB randomly generated dataset. As it is illustrated in the figure, each node of DRESS completes its iSAX summarization phase on its 50GB chunk in almost half the time, i.e., 4.4 seconds, compared to (the single-node) MESSI (on the 100GB dataset) that requires 8.7 seconds. This is so, since each node of DRESS works on the half dataset and its iSAX summarization workload is massively parallel, i.e., contains zero communication (over the nodes) and almost zero synchronization (among the threads of a specific node) overhead. This is the best that DRESS can achieve and is observed when a) each of MESSI-1st-half and MESSI-2nd-half completes its iSAX summarization phase on its 50GB chunk in the same amount of time, and b) the two nodes of DRESS require identical amount of time (i.e., 4.4 seconds) to complete their processing. Therefore, we see that DRESS achieves 2x speedup (over MESSI) and its two nodes do not experience workload imbalance problems in this case.

Similarly, our results for the **tree index construction phase** Figure *7*-b, shows that, again, DRESS achieves almost the best possible speedup. This is so, since DRESS requires almost the same amount of time to process the whole 100GB dataset with the amount of time that each of MESSI-1st-half and MESSI-2nd-half requires to process half of it. Comparing the performance of the two nodes of DRESS (i.e., the orange bars in Figure *7* -b), we find out that the second node requires 5% more time than the first one, concluding that DRESS does not experience a big workload imbalance for its tree index construction phase.

Moreover, we observe that the two nodes of DRESS achieve speedup more than 2x compared to MESSI. To better understand this, we studied the heights of the tree indexes constructed by these algorithms. Since each node of DRESS builds its index tree only on a half of the dataset (i.e., on a 50GB chunk), we expect the height of each of the tree indexes to be reduced. Indeed, this is the case, the height of the tree index constructed by MESSI (for the 100GB dataset) is 22, while each node of DRESS constructs (for its 50GB chunk) a tree index of height 20 respectively. Therefore, DRESS achieves more than 2x speedup, since additionally to working on the half dataset, each of its nodes inserts elements into tree indexes of smaller height.

Finally, Figure 7-c, shows the performance of the **query answering phase** of both the above algorithms and DRESS-SW-BSF. DRESS achieves less than 2x speedup (that would be the ideal) compared to MESSI. However, once more, DRESS requires almost the same amount of time to answer queries on the entire dataset with the amount of time that each of MESSI-1st-half and MESSI-2nd-half requires to answer queries on half of the dataset. Therefore, DRESS inherits this performance overhead from MESSI.

Additionally, we observe that the first node of DRESS (and MESSI-1st-half) is almost 2.5% slower than its second node (and than MESSI-2nd-half). To better understand this, we measured the number of the calculated lower bound distances (Figure *7*-d) and the average number of real distances (Figure *7*-e) that each DRESS node calculates. Our results show that the observed behavior is due to the fact that the first node of DRESS performs ~3.3% more lower bound distances and ~7.7% more real distances than the second node of DRESS. Since the nodes of DRESS work on chunks of randomly generated data, we speculate that the reason for these observed differences is that the second node may achieve better tree index pruning and/or discover better answers (i.e., answers with smaller distances) and thus, process less priority queues elements. In any case, we conclude that DRESS experiences low workload imbalance between its two nodes, in the random dataset case. Although the resulting performance overhead is insignificant, we tried to alleviate it by introducing the *SW-BSF* technique (i.e., DRESS-SW-BSF).

Interestingly, Figure *7*-c, shows also that each node of DRESS-SW-BSF (blue bars) achieves more than 5% better performance than each node of DRESS (and MESSI-1st-half and MESSI-2nd-half, respectively) and this is again due to that DRESS-SW-BSF calculates less lower bound and real distances (Figure *7*-d and Figure *7*-e respectively). We remind that DRESS-SW-BSF is similar to DRESS, where additionally each node broadcasts its initial BSF and any update of its BSF value, to the other node. So, by doing this, we expected that each node of DRESS-SW-BSF would achieve both better pruning during the tree index construction and process less priority queues elements. Our results confirm our expectations. We remark that we only present our results for the query answering phase of DRESS-SW-BSF, since the results for its other two phases (i.e., iSAX summarization and tree index construction) are similar to the ones of DRESS.

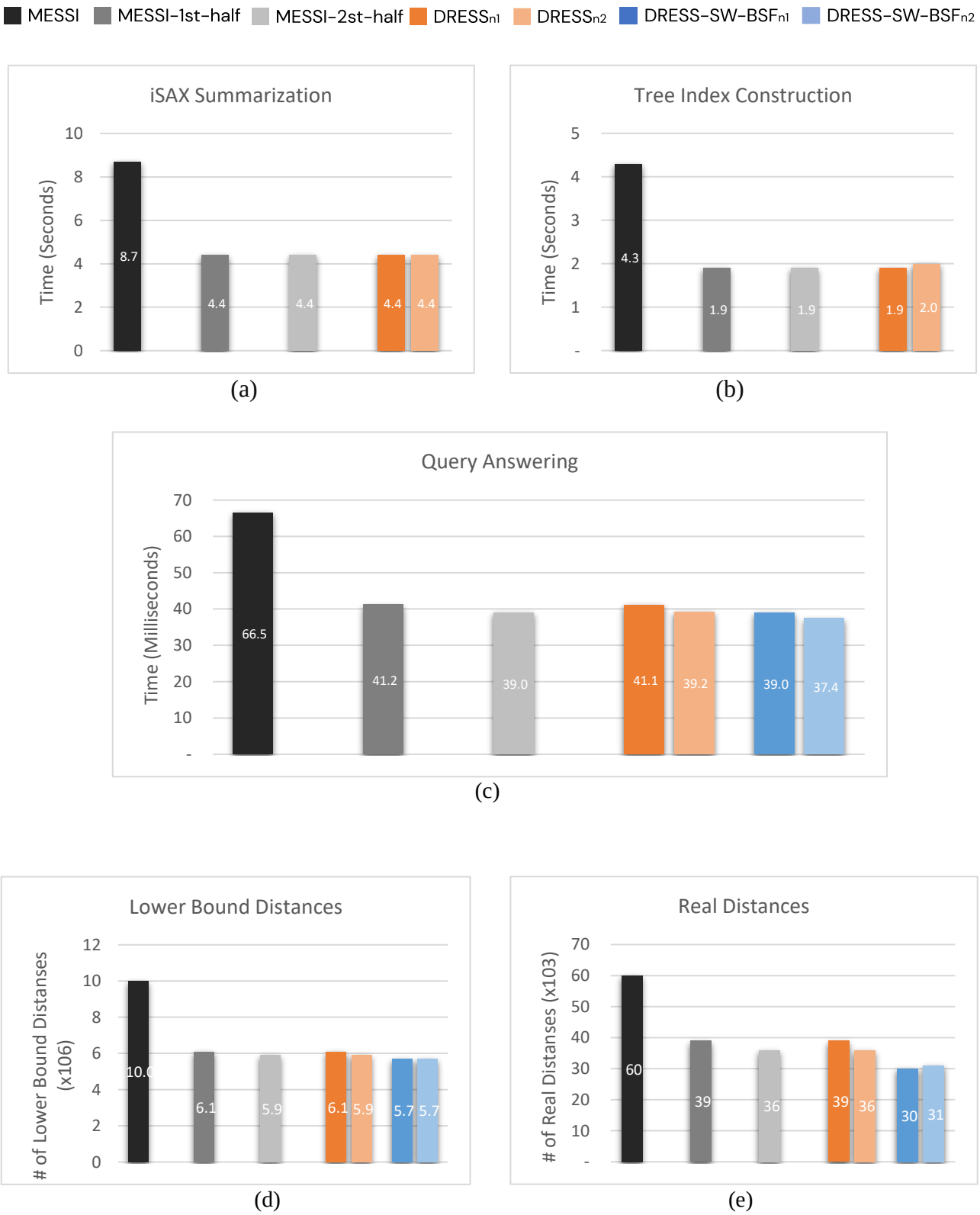


Figure 7 - Random Generated Dataset

6.3.2 Seismic dataset

presents our results for the iSAX summarization and the tree index construction phases of DRESS and MESSI on the 100GB Seismic dataset, and MESSI-1st-half and MESSI-2nd-half on each of the

two 50GB chunks of the 100GB Seismic dataset, respectively. We observe similar trends with our results for the randomly generated dataset, concluding again that DRESS does not experience workload imbalance problems, in these two phases. Specifically, for the iSAX summarization phase, we conclude that in this case DRESS achieves almost 2x speedup (over MESSI) and its two nodes do not experience workload imbalance problems.

Regarding the tree index construction phase, DRESS achieves more than 2x (and up to 3x, for the second node) speedup; this is again due to the heights of the tree constructed tree indexes. Specifically, for the seismic dataset, the height of the tree index constructed by MESSI (for the 100GB dataset) is 50, while each node of DRESS constructs (for its 50GB chunk) a tree index with height 45 and 29, respectively. Notice that the observed tree indexes heights' difference for the second node compared to MESSI (i.e., 50-29=21) is much bigger that the corresponding difference in the randomly generated dataset (i.e., 22-20=2). This justifies the observed 3x speedup for the tree index construction phase of the second node. Moreover, comparing the performance of the two nodes of DRESS (i.e., the orange bars in -b), the first node requires almost 33% more time than the second one, concluding that DRESS experiences moderate workload imbalance for its tree index construction phase. To better understand this, we counted the empty root subtrees of each node's tree index. Interestingly, the empty root subtrees of the tree index of the first node are many more than the ones of the tree index of the second node; specifically, 400 and 123, respectively. Since both nodes process the same amount of data series, it follows that the root subtrees of the tree index of the second node should have less height than the root subtrees of the first nodes. We justified this by counting the height of each root subtree and as expected, the average height of the second node's root subtrees is 1.6, while for the first one is 1.9. Combining this, with the fact that the total heights of their tree indexes are 45 and 29, respectively, it follows that the second node performs much less work to populate its tree index. However, this is only an indication for the observed behavior. Due to lack of time, we have not elaborated more on this issue to better understand it and/or try to fix it, since we focused (and devoted a large amount of working time) on understanding and optimizing the surprising results of the query answering phase of DRESS, as described in the following. Nevertheless, notice that the tree index construction time of DRESS (and MESSI) on Seismic dataset (-b) is more that 2x faster than its tree index construction time on the randomly generated dataset (Figure *7*-b). Comparing the empty root subtrees of the two tree indexes of DRESS (and MESSI) for these two datasets, we observe that the one for the randomly generated dataset has over 19x more empty root subtrees than the one for the Seismic dataset. This observation is another indication for the observed behavior of DRESS (and MESSI) on these two datasets, and it additionally strengthens our previous reasoning concerning the performance difference of the tree index construction phases among the two nodes of DRESS.

Finally, -c, shows the performance of the query answering phase of the above algorithms, DRESS-shuffled, and DRESS-SW-BSF. Interestingly, the two nodes of DRESS experience great workload imbalance. Specifically, the first node completes its query answering phase in almost half time, while the second node surprisingly completed in 4x larger amount of time, compared to MESSI (that is working on the while 100GB dataset); recall that each node of DRESS works only on its own 50GB chunk. However, once more, each node of DRESS requires almost the same amount of time to answer the queries on the whole 100GB dataset with the amount of time that each of MESSI-1st-half and MESSI-2nd-half requires to answer the queries on half of it, respectively. Therefore, DRESS inherits this behavior from MESSI, and it is not due to its distributed implementation.

To better understand this, we measured the numbers of the calculated lower bound (– d) and real (-e) distances. Our results show that the observed behavior is due to the fact that the second node of DRESS (and MESSI-2nd-half) performs 2.5x more lower bound distances and more than 8.3x more real distances than the second node of DRESS (and MESSI-1st-half). So, it seems that the first chunk of Seismic contains data series that are closely related to the queries, while the second chunk contains only data series that are not in close distance to the queries and a lot of them are in similar distance from the queries; the latter justifies the huge number of lower bound and real distance calculations performed by the second node. These are possible, given that the nodes of DRESS work on chunks of Seismic, which is a dataset that contains real data series (and not randomly generated ones). Therefore, their characteristics may vary significantly among the (two) chunks.

To examine this, we chose to shuffle the Seismic dataset so that the in close distance (to the queries) data series that are located in the first chunk of Seismic, are scattered among the two chunks of the shuffled Seismic dataset, i.e., we experimented with DRESS-shuffled. Our results show that the nodes of DRESS-shuffled complete in almost the same amount of time, which means that DRESS-shuffled alleviated the workload imbalance problem of DRESS. However, although DRESS-shuffled achieves better performance than DRESS, it is still 2x slower than MESSI. To understand this surprising behavior, we studied the time required for DRESS (Figure 9-a) and DRESS-shuffled (Figure 9-b) to answer each one of the queries (the averages of which are reported in - c). While in DRESS the first node constantly completes faster than the second node, in DRESS-shuffled one after the other each one of the two nodes completes faster, i.e., in an alternate way. Therefore, we conclude that each query has only an in close distance data series in Seismic. Our results revealed that shuffling the Seismic dataset only scatters these in close distance data series among the chunks. So, each chunk in DRESS-shuffled has an in close distance data series for half of the queries, with high probability. Thus, each node performs a small amount of work for half queries and a large amount of work for the other half, resulting on (almost) the same average answering time for both nodes, while per query the two nodes experience workload imbalance. This is also justified by the lower bound and real distance calculations shown in -d and -e.Interestingly, in such cases, where the dataset contains only a single in close distance data series (for each query), shuffling the dataset is unable to solve the workload imbalance problem for each single query. Furthermore, the system may even experience performance degradation, which is what happened in our case.

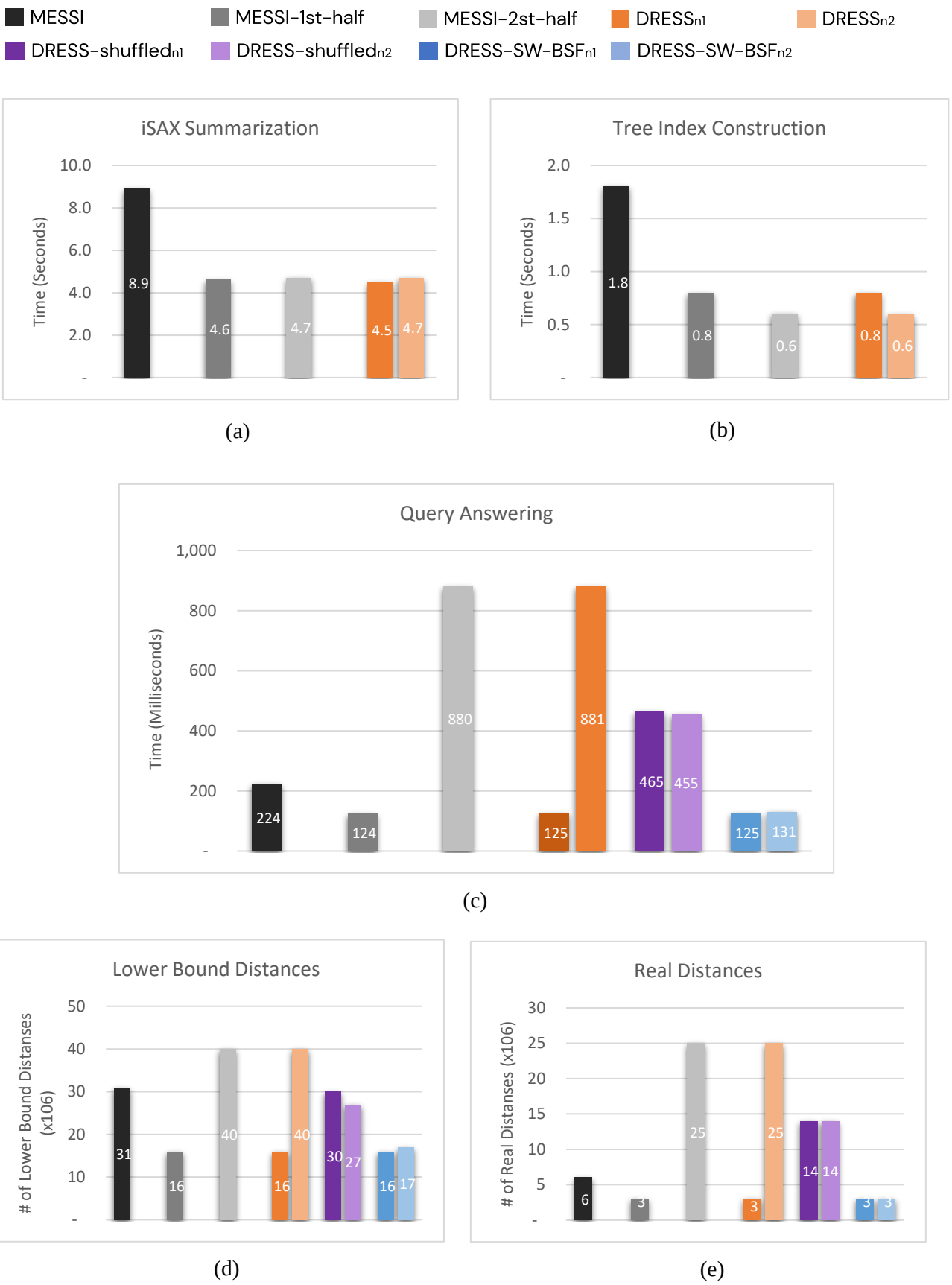
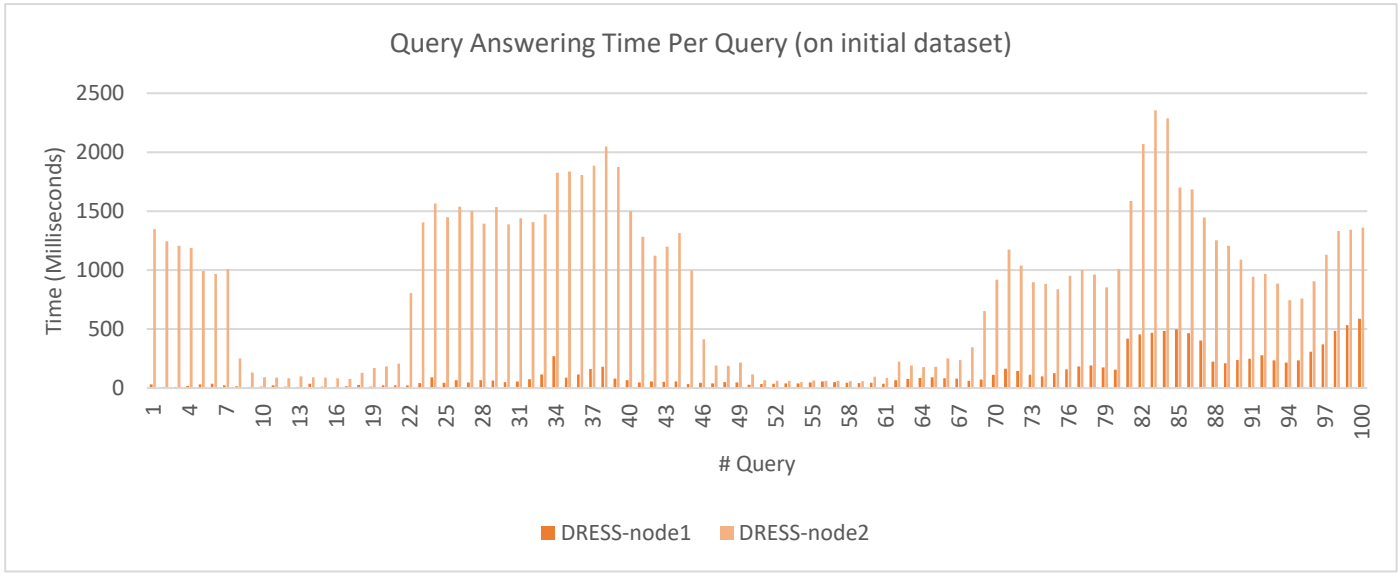
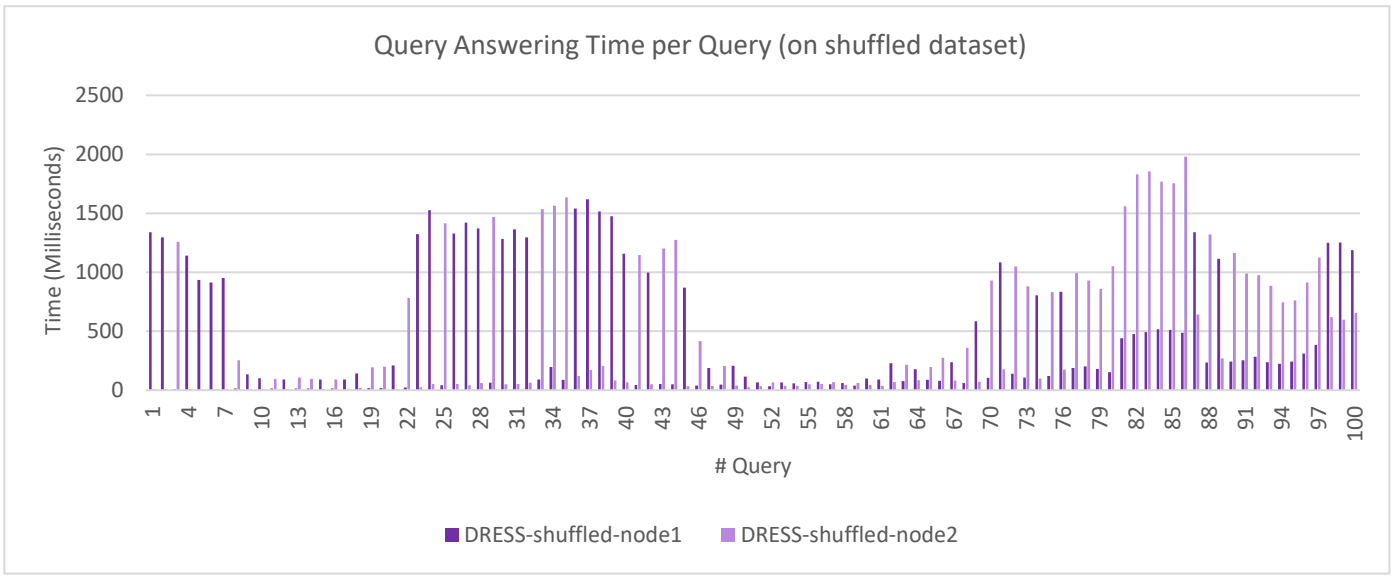


Figure 8 - Seismic Dataset



(a)



(b)

Figure 9 - Seismic Dataset Query Answering Time Per Query

To solve the workload imbalance problem for each single query, we introduced the SW-BSF technique (i.e., DRESS-SW-BSF). - c, shows that the nodes of DRESS-SW-BSF complete their query answering phases (for all the queries) in almost the same amount of time (with the second node being 4.8% slower) and it is 1.7x faster than MESSI. Therefore, DRESS-SW-BSF successfully achieves speedup close to the ideal (i.e., 2x, given that our testing system incorporates two nodes), with negligible workload imbalance. This is also justified by the lower bound and real distance calculations shown in -d and – e.

We remark that we only present our results for the query answering phase of DRESS-shuffled and DRESS-SW-BSF, since the results for the rest of their phases (i.e., iSAX summarization and tree index construction) are similar to the ones of DRESS.

6.3.3 Work stealing

As we have already discussed in Section 4.3, there are cases where the BSF may not work well enough since a node may remain idle for a long period of time waiting for the other node to complete its work. Additionally, as we have already discussed in Section 4.2, shuffling the dataset may not be applicable in some real-time environments, where response time is crucial; this may again result in workload imbalance. In these cases, to achieve better workload balance, work-stealing techniques can be applied. Therefore, we experimented with DRESS-work-steal.

Specifically, Figure 10-a and Figure 10-b present our results for the iSAX summarization and the tree index construction phases of DRESS, DRESS-work-steal, and MESSI on the 100GB Seismic dataset. We can see that DRESS-work-steal does not experience workload imbalance problems in both phases. Moreover, it completes its summarization phase 2x slower than DRESS and its tree index construction almost 2.2x slower. This is so, since DRESS-work-steal summarizes the whole 100GB Seismic dataset and produces both tree indexes, one for each 50GB chunk of it, so that it is able to apply the work stealing technique on tree indexes (described in Section 4.4.2). Notice that DRESS-work-steal completes these phases in exactly the same period of time with MESSI, as expected. Additionally, Figure 10-c shows the performance of the query answering phase of the above algorithms and DRESS-shuffled. We can see that the two nodes of DRESS-work-steal experience no workload imbalance problems (thus, DRESS-work-steal achieves its goal), while its performance is almost 2.4x times slower than MESSI. This is so, since as already explained in Section 6.3.2, the characteristics of the two chunks of Seismic vary significantly, with the queries being closely related to only a single data series of the first chunk. Therefore, the first node completes its query answering phase (for each query) faster (than the second node) and continues by stealing work from the second node. However, given that the second node performs unnecessary work (since all its data series aren't in close distance to the query), it follows that the first node also performs unnecessary work. Due to this, both of them experience performance degradation (compared to MESSI). Moreover, DRESS-work-steal is 1.15x slower than DRESS-shuffled. Although both these techniques attempt to alleviate the workload imbalance problem that DRESS experiences, the work stealing technique of DRESS-work-steal is applied during the critical

path of the query answering phase, while the shuffling technique of DRESS-shuffled is applied outside this critical path.

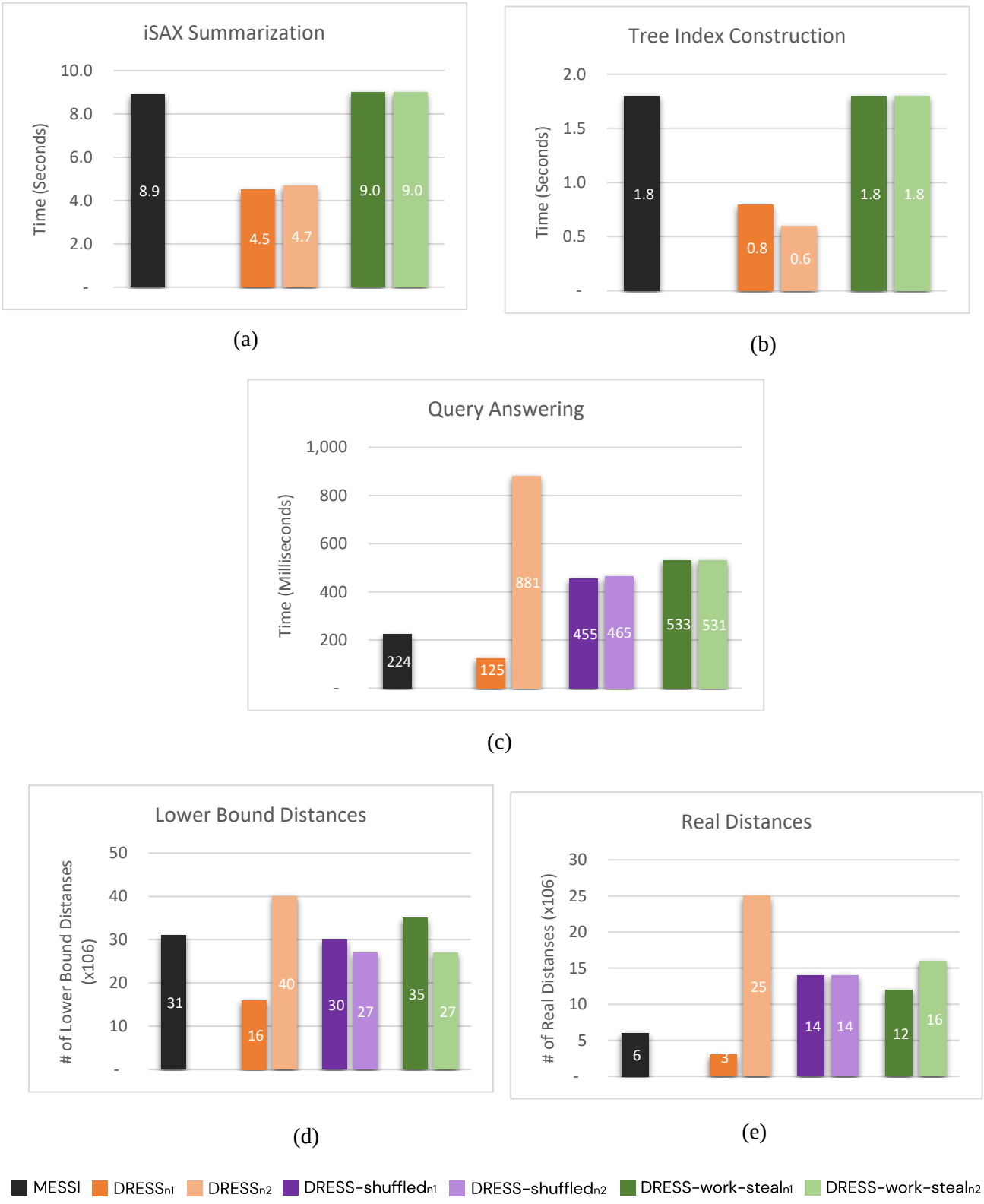


Figure 10 - Seismic Work-Stealing

Using this as an indication for the observed behavior, recall that in order to implement DRESS-work-steal we have changed the tree index pruning procedure. Specifically, as described in Section 4.4.2, each node performs the pruning of its tree index into parts, so that the other node can steal a part of it. However, this change affects the effectiveness of the priority queues since at each step they are populated with the candidate (for real distance computation) leaf nodes of only a specific part of the tree index, as opposed to the complete tree index of DRESS. This may result in less relevant candidate leaf nodes at the head of the priority queues. Due to this, DRESS-work-steal calculates more real distances than DRESS-shuffled. To investigate this, we measured the lower bound and the real distance calculations performed by DRESS-work-steal and DRESS-shuffled, as figures 10-d and 10-e depicts. Interestingly, the lower bound and the real distance calculations performed by the first node of DRESS-work-steal are more than the ones performed by the first node of DRESS; although its chunk contain the final answer and thus, someone could expect that its calculations should not be greatly affected. However, this is not a correct expectation, for a reason similar to the above, as follows. Recall that we produce the 100 queries on Seismic dataset by incrementally adding noise to the first 100 data series. Notice that while increasing the noise added to a query, the probability of calculating an initial BSF that is not in close distance to the final answer also increases. Combining this with the fact that DRESS-work-steal works in parts of the tree index, it follows that until the first node finds the final answer, the probability of performing more lower bound and real distance calculations (in comparison to when answering the same query with no added noise) increases. This is what we observer is our experiments.

6.3.4 Parallel queries on Seismic dataset

Considering a system with independent queries that are provided in batches by the external environment, we implemented DRESS-parallel-queries that incorporates the parallel execution of multiple queries technique (described in Section 4.5). Specifically, we partitioned the 100 queries into two parts of 50 elements, with the first part containing the even queries and the second the odd ones. Then, each of these (two) parts is assigned to n_1 and n_2, respectively. We compared DRESS-parallel-queries with MESSI. Recall that to process its assigned queries on its own, each node of DRESS-parallel-queries has locally replicated the whole (100GB) dataset, similarly to (single node) MESSI. Therefore, we only present our results for its query answering phase, since the rest of its phases (i.e., iSAX summarization and tree index construction) are completed by each node of DRESS-parallel-queries at the same time with MESSI; thus, these results are not interesting.

Figure 11-a depicts the execution time of the query answering phase of the tested algorithms. DRESS-parallel-queries achieves speedup of 1.8, compared to MESSI, and the two nodes of DRESS-parallel-queries experience small workload imbalance. To better understand this, we counted the lower bound and the real distances calculated by the nodes of DRESS-parallel-queries. The results are presented in Figure 11-b and Figure 11-c, where the same workload imbalance between the

two nodes is again present. So, this imbalance does not come due to the distributed implementation of DRESS-parallel-queries, but due to the initial partitioning of the queries.

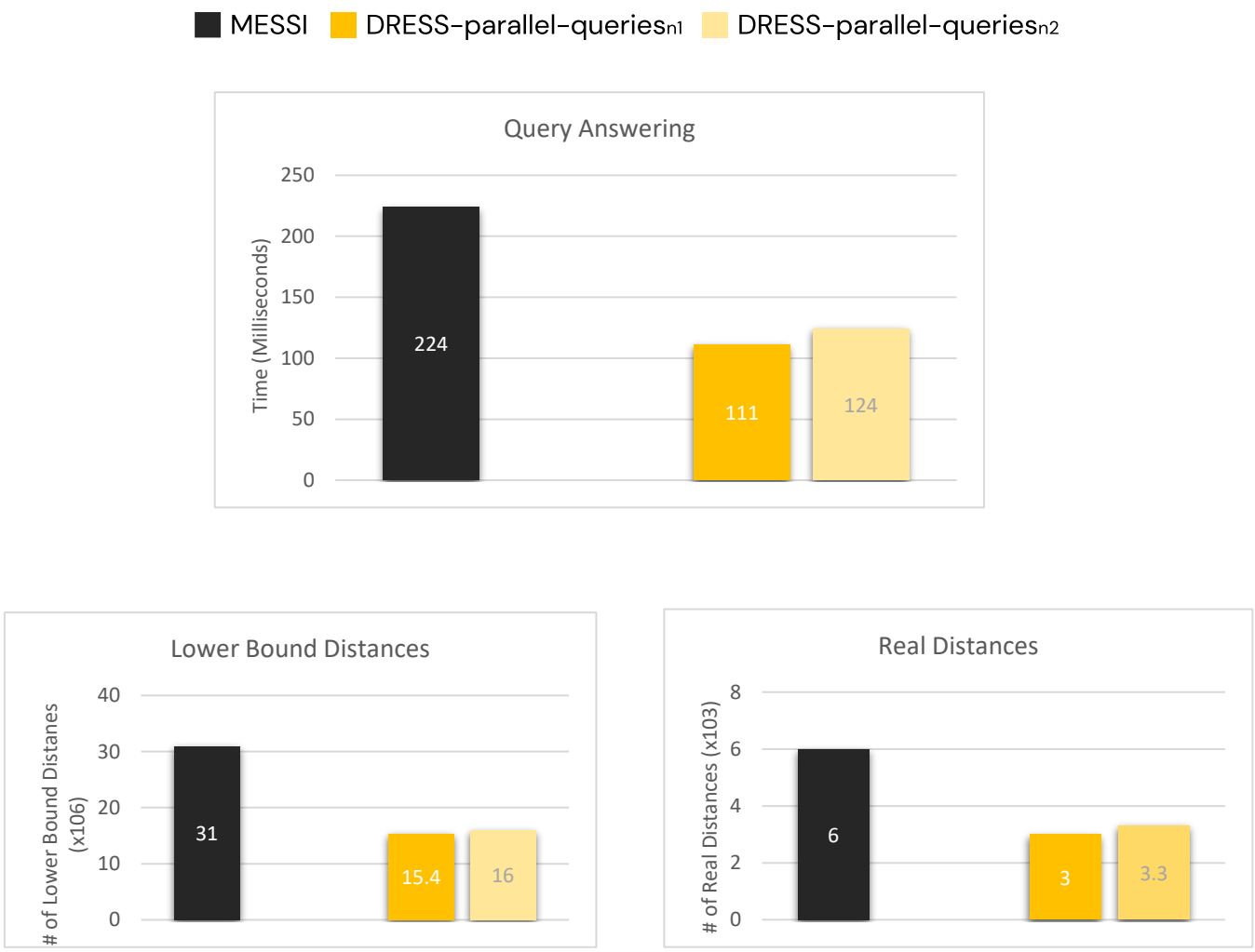


Figure 11 - Seismic Parallel Queries

7. Conclusions

In this deliverable, we present fundamental techniques for distributed query processing of big collections of data series in a system of multiple nodes. Deliverable 4.1 builds upon the results here and significantly extends them to come up with the integrated data series index of PLATON, which exploits the full power of distributed systems with a big number of nodes.

In a different but related avenue, in [35], we present Hercules, a parallel tree-based technique for exact similarity search on massive disk-based data series collections. The work in [35] presents novel index construction and query answering algorithms that leverage different summarization techniques, carefully schedule costly operations, optimize memory and disk accesses, and exploit the multi-threading and SIMD capabilities of modern hardware to perform CPU-intensive calculations. The work demonstrates the superiority and robustness of Hercules with an extensive experimental evaluation against the state-of-the-art techniques, using a variety of synthetic and real datasets, and query workloads of varying difficulty. The results show that Hercules performs up to one order of magnitude faster than the best competitor (which is not always the same). Moreover, Hercules is the only index that outperforms the optimized scan on all scenarios, including the hard query workloads on disk-based datasets.

Acknowledgments: This work has been performed while P. Fatourou was working at the LIPADE, Université Paris Cité, as an MSCA Individual Fellow in the context of the PLATON project (MSCA grant agreement #101031688).

8. References

[1] B. Peng, T. Palpanas, and P. Fatourou, “MESSI: In-Memory Data Series Indexing”

[2] R. Agrawal, C. Faloutsos, and A. N. Swami, “Efficient similarity search in sequence databases,” in FODO, 1993.

[3] T. Rakthanmanon, B. J. L. Campana, A. Mueen, G. E. A. P. A. Batista, M. B. Westover, Q. Zhu, J. Zakaria, and E. J. Keogh, “Searching and mining trillions of time series subsequences under dynamic time warping,” in SIGKDD, 2012.

[4] A. Guillaume, “Head of Operational Intelligence Department Airbus. Personal communication.” 2017.

[5] Kostas Zoumpatianos, Stratos Idreos, and Themis Palpanas, “ADS: The Adaptive Data Series Index”.

[6] B. Peng, T. Palpanas, and P. Fatourou, “ParIS: The Next Destination for Fast Data Series Indexing and Query Answering”.

[7] B. Peng, T. Palpanas, and P. Fatourou, “ParIS+: Data Series Indexing on Multi-Core Architectures.” 2021.

[8] Pablo Huijse, Pablo A. Estevez, Pavlos Protopapas, Jose C. ´ Principe, and Pablo Zegers. Computational intelligence challenges and applications on large-scale astronomical time-series databases. IEEE Comp. Int. Mag., 9(3):27–39, 2014.

[9] Kunio Kashino, Gavin Smith, and Hiroshi Murase. Time-series active search for quick retrieval of audio and video. In ICASSP, 1999.

[10] Michele Linardi and Themis Palpanas. ULISSE: ultra-compact index for variable-length similarity search in data series. In ICDE, 2018.

[11] Michele Linardi, Yan Zhu, Themis Palpanas, and Eamonn J. Keogh. VALMOD: A suite for easy and exact detection of variable length motifs in data series. In SIGMOD, 2018.

[12] Themis Palpanas. Data series management: The road to big sequence analytics. SIGMOD Record, 44(2):47–52, 2015.

[13] Michele Linardi, Yan Zhu, Themis Palpanas, and Eamonn J. Keogh. Matrix profile X: VALMOD - scalable discovery of variable-length motifs in data series. In SIGMOD, 2018.

[14] T. Rakthanmanon, B. Campana, A. Mueen, G. Batista, B. Westover, Q. Zhu, J. Zakaria, and E. Keogh. Searching and mining trillions of time series subsequences under dynamic time warping. In KDD, 2012.

[15] Kostas Zoumpatianos and Themis Palpanas. Data series management: Fulfilling the need for big sequence analytics. In ICDE, 2018.

[16] Karima Echihabi, Kostas Zoumpatianos, Themis Palpanas, and Houda Benbrahim. The lernaean hydra of data series similarity search: An experimental evaluation of the state of the art. PVLDB, 2019

[17] Philippe Esling and Carlos Agon. Time-series data mining. ACM Comput. Surv., 45(1):12:1–12:34, December 2012.

[18] Ira Assent, Ralph Krieger, Farzad Afschari, and Thomas Seidl. The ts-tree: Efficient time series search and retrieval. In EDBT, 2008.

[19] A. Camerra, J. Shieh, T. Palpanas, T. Rakthanmanon, and E. J. Keogh. Beyond one billion time series: indexing and mining very large time series collections with iSAX2+. Knowl. Inf. Syst., 2014.

[20] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. Fast subsequence matching in time-series databases. SigRec, 23(2):419–429, 1994

[21] Themis Palpanas. The parallel and distributed future of data series mining. In International Conference on High-Performance Computing & Simulation, HPCS, 2017.

[22] Botao Peng, Themis Palpanas, and Panagiota Fatourou. ParIS: The Next Destination for Fast Data Series Indexing and Query Answering. IEEE BigData, 2018.

[23] Djame Edine Yagoubi1 , RezaAkbarinia , Florent Masseglia , Themis Palpanas. DPiSAX: Massively Distributed Time Series Indexing and Querying.

[24] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, Ion Stoica. Spark: Cluster Computing with Working Sets

[25] Anshu Kumari, MPI: A message-passing interface

[26] J. Shieh and E. Keogh, “i sax: indexing and mining terabyte sized time series,” in SIGKDD, 2008.

[27] “Incorporated Research Institutions for Seismology – Seismic Data Access,” http://ds.iris.edu/data/access 2016.

[28] K. Zoumpatianos and T. Palpanas, “Data series management: Fulfilling the need for big sequence analytics,” in ICDE, 2018

[29] Themis Palpanas. Big sequence management: A glimpse of the past, the present, and the future. In SOFSEM, 2016.

[30] Themis Palpanas. The parallel and distributed future of data series mining. In International Conference on High Performance Computing & Simulation, HPCS, 2017

[31] Djamel-Edine Yagoubi, RezaAkbarinia, Florent Masseglia and Themis Palpanas. DPiSAX: Massively Distributed Partitioned iSAX

[32] B. Peng, T. Palpanas, and P. Fatourou, “SING: Sequence Indexing Using GPUs”

[33] B. Peng, T. Palpanas, and P. Fatourou, “Paris: The next destination for fast data series indexing and query answering,” IEEE BigData, 2018.

[34] B. Peng, T. Palpanas, and P. Fatourou, “Paris+: Data series indexing on multi-core architectures,” TKDE, 2020.

[35] Karima Echihabi, Panagiota Fatourou, Kostas Zoumpatianos, Themis Palpanas, Houda Benbrahim, ``Hercules Against Data Series Similarity Search,” submitted for publication. LIPADE-TR-No 7, October 2022.